DEEPMATH

MATHÉMATIQUES (SIMPLES) DES RÉSEAUX DE NEURONES (PAS TROP COMPLIQUÉS)

ARNAUD BODIN & FRANÇOIS RECHER

ALGORITHMES ET MATHÉMATIQUES



Mathématiques des réseaux de neurones

Introduction

Ce livre comporte deux parties avec pour chacune un côté « mathématique » et un côté « réseau de neurones » :

- analyse et réseaux de neurones,
- algèbre et convolution.

Le but de la première partie est de comprendre les mathématiques liées aux réseaux de neurones et le calcul des poids par rétropropagation. La seconde est consacrée à la convolution qui est une opération mathématique simple pour extraire des caractéristiques d'une image et permet d'obtenir des réseaux de neurones performants.

Nous limitons les mathématiques présentées au niveau de la première année d'études supérieures, ce qui permet de comprendre les calculs de la rétropropagation. Ce livre explique comment utiliser des réseaux de neurones simples (à l'aide de *tensorflow*/*keras*). À l'issue de sa lecture, vous saurez programmer un réseau qui distingue un chat d'un chien ! Le lecteur est supposé être familier avec les mathématiques du niveau lycée (dérivée, étude de fonction…) et avec les notions de base de la programmation avec *Python*.

Selon votre profil vous pouvez suivre différents parcours de lecture :

- le *novice* étudiera le livre dans l'ordre, les chapitres alternant théorie et pratique. Le danger est de se perdre dans les premiers chapitres préparatoires et de ne pas arriver jusqu'au cœur du livre.
- le *curieux* picorera les chapitres selon ses intérêts. Nous vous conseillons alors d'attaquer directement par le chapitre « Réseau de neurones » puis de revenir en arrière pour revoir les notions nécessaires.
- le *matheux* qui maîtrise déjà les fonctions de plusieurs variables pourra commencer par approfondir ses connaissances de *Python* avec *numpy* et *matplotlib* et pourra ensuite aborder *tensorflow*/*keras* sans douleurs. Il faudra cependant comprendre la « dérivation automatique ».
- l'*informaticien* aura peut-être besoin de revoir les notions de mathématiques, y compris les fonctions d'une variable qui fournissent un socle solide, avant d'attaquer les fonctions de deux variables ou plus.

Ce cours est aussi disponible en vidéos « Youtube : Deepmath ».
L'intégralité des codes *Python* ainsi que tous les fichiers sources sont sur la page *GitHub* d'Exo7 :
« GitHub : Exo7 ».

Sommaire

Résumé des chapitres

Dérivée

La notion de dérivée joue un rôle clé dans l'étude des fonctions. Elle permet de déterminer les variations d'une fonction et de trouver ses extremums. Une formule fondamentale pour la suite sera la formule de la dérivée d'une fonction composée.

Python : numpy et matplotlib avec une variable

Le but de ce court chapitre est d'avoir un aperçu de deux modules *Python* : *numpy* et *matplotlib*. Le module *numpy* aide à effectuer des calculs numériques efficacement. Le module *matplotlib* permet de tracer des graphiques.

Fonctions de plusieurs variables

Dans ce chapitre, nous allons nous concentrer sur les fonctions de deux variables et la visualisation de leur graphe et de leurs lignes de niveau. La compréhension géométrique des fonctions de deux variables est fondamentale pour assimiler les techniques qui seront rencontrées plus tard avec un plus grand nombre de variables.

Python : numpy et matplotlib avec deux variables

Le but de ce chapitre est d'approfondir notre connaissance de *numpy* et *matplotlib* en passant à la dimension 2. Nous allons introduire les tableaux à double entrée qui sont comme des matrices et visualiser les fonctions de deux variables.

Réseau de neurones

Le cerveau humain est composé de plus de 80 milliards de neurones. Chaque neurone reçoit des signaux électriques d'autres neurones et réagit en envoyant un nouveau signal à ses neurones voisins. Nous allons construire des réseaux de neurones artificiels. Dans ce chapitre, nous ne chercherons pas à expliciter une manière de déterminer dynamiquement les paramètres du réseau de neurones, ceux-ci seront fixés ou bien calculés à la main.

Python : tensorflow avec keras - partie 1

Le module *Python tensorflow* est très puissant pour l'apprentissage automatique. Le module *keras* a été élaboré pour pouvoir utiliser *tensorflow* plus simplement. Dans cette partie nous continuons la partie facile : comment utiliser un réseau de neurones déjà paramétré ?

Gradient

Le gradient est un vecteur qui remplace la notion de dérivée pour les fonctions de plusieurs variables. On sait que la dérivée permet de décider si une fonction est croissante ou décroissante. De même, le vecteur gradient indique la direction dans laquelle la fonction croît ou décroît le plus vite. Nous allons voir comment calculer de façon algorithmique le gradient grâce à la « différentiation automatique ».

Descente de gradient

L'objectif de la méthode de descente de gradient est de trouver un minimum d'une fonction de plusieurs variables le plus rapidement possible. L'idée est très simple, on sait que le vecteur opposé au gradient indique une direction vers des plus petites valeurs de la fonction, il suffit donc de suivre d'un pas cette direction et de recommencer. Cependant, afin d'être encore plus rapide, il est possible d'ajouter plusieurs paramètres qui demandent pas mal d'ingénierie pour être bien choisis.

Rétropropagation

La rétropropagation, c'est la descente de gradient appliquée aux réseaux de neurones. Nous allons étudier des problèmes variés et analyser les solutions produites par des réseaux de neurones.

Python : tensorflow avec keras - partie 2

Jusqu'ici nous avons travaillé dur pour comprendre en détails la rétropropagation du gradient. Les exemples que nous avons vus reposaient essentiellement sur des réseaux simples. En complément des illustrations mathématiques étudiées, il est temps de découvrir des exemples de la vie courante comme la reconnaissance d'image ou de texte. Nous profitons de la librairie *tensorflow*/*keras* qui en quelques lignes nous permet d'importer des données, de construire un réseau de neurones à plusieurs couches, d'effectuer une descente de gradient et de valider les résultats.

Convolution : une dimension

Ce chapitre permet de comprendre la convolution dans le cas le plus simple d'un tableau à une seule dimension.

Convolution

La convolution est une opération mathématique simple sur un tableau de nombres, une matrice ou encore une image afin d'y apporter une transformation ou d'en tirer des caractéristiques principales.

Convolution avec Python

Python permet de calculer facilement les produits de convolution.

Convolution avec tensorflow/keras

Nous mettons en œuvre ce qui a été vu dans les chapitres précédents au sujet des couches de convolution afin de créer des réseaux de neurones beaucoup plus performants.

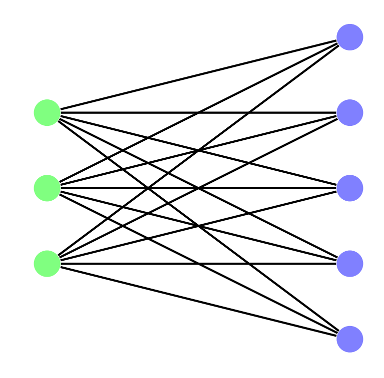
Tenseurs

Un tenseur est un tableau à plusieurs dimensions, qui généralise la notion de matrice et de vecteur et permet de faire les calculs dans les réseaux de neurones.

Probabilités

Nous présentons quelques thèmes probabilistes qui interviennent dans les réseaux de neurones.

Première partie



ANALYSE – RÉSEAUX DE NEURONES

Chapitre 1

Dérivée

Vidéo ■ partie 1.1. Définition de la dérivée
Vidéo ■ partie 1.2. Dérivée d'une composition
Vidéo ■ partie 1.3. Dérivation automatique
Vidéo ■ partie 1.4. Fonctions d'activation
Vidéo ■ partie 1.5. Minimums et maximums

La notion de dérivée joue un rôle clé dans l'étude des fonctions. Elle permet de déterminer les variations d'une fonction et de trouver ses extremums. Une formule fondamentale pour la suite sera la formule de la dérivée d'une fonction composée.

Ceux qui sont à l'aise en mathématiques peuvent se rendre directement à la deuxième section de ce chapitre consacrée à la « dérivation automatique ».

1. Dérivée

1.1. Définition

Soit $f : I \to \mathbb{R}$ une fonction, où I est un intervalle ouvert de $\mathbb{R}$ (par exemple du type $]a, b[$). Soit $x_0 \in I$.

Définition.
La dérivée de f en x_0, si elle existe, est le nombre

$$\boxed{f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}.}$$

C'est donc la limite du taux d'accroissement $\frac{f(x)-f(x_0)}{x-x_0}$ lorsque x tend vers x_0. Nous noterons la dérivée de f en x_0 indifféremment sous la forme :

$$\boxed{f'(x_0) \qquad \text{ou} \qquad \frac{\mathrm{d}f}{\mathrm{d}x}(x_0).}$$

Remarque.
Une dérivée n'existe pas toujours. Dans ce cours nous supposerons par défaut que la dérivée est bien définie, c'est-à-dire que f est ***dérivable*** en x_0. Nous préciserons explicitement les situations pour lesquelles ce n'est pas le cas. Comme la limite est unique, la dérivée de f en x_0 ne peut prendre qu'une seule valeur.

Exemple.
Calculons la dérivée en $x_0 = 1$ de la fonction f définie par $f(x) = x^2$. On commence par réécrire le taux

d'accroissement :
$$\frac{f(x)-f(1)}{x-1}=\frac{x^2-1}{x-1}=\frac{(x-1)(x+1)}{x-1}=x+1.$$
Ce taux d'accroissement tend vers 2 lorsque x tend vers 1, donc $f'(1)=2$.
Plus généralement, on montre que $f'(x_0)=2x_0$:
$$\frac{f(x)-f(x_0)}{x-x_0}=\frac{x^2-x_0^2}{x-x_0}=\frac{(x-x_0)(x+x_0)}{x-x_0}=x+x_0\xrightarrow[x\to x_0]{}2x_0.$$

Exemple.
On connaît la limite suivante :
$$\frac{\exp(x)-1}{x}\xrightarrow[x\to 0]{}1.$$
Interprétons ceci en termes de dérivée. Soit $f(x)=\exp(x)$. Alors la limite ci-dessus s'écrit :
$$\frac{f(x)-f(0)}{x-0}\xrightarrow[x\to 0]{}1,$$
c'est-à-dire $f'(0)=1$. Autrement dit, la dérivée de l'exponentielle en 0 vaut 1.

Pour chaque x_0 en lequel la fonction f est dérivable, on associe un nombre $f'(x_0)$, ce qui nous permet de définir une nouvelle fonction.

Définition.
La fonction qui à x associe $f'(x)$ est la ***fonction dérivée*** de f. On la notera de l'une des façons suivantes :
$$x\mapsto f'(x)\qquad\text{ou}\qquad f'\qquad\text{ou}\qquad\frac{\mathrm{d}f}{\mathrm{d}x}.$$
Pour le premier exemple avec $f(x)=x^2$, nous avons montré que $f'(x)=2x$ (et donc $f'(1)=2$). L'exponentielle possède la propriété fondamentale que sa dérivée est aussi l'exponentielle : si $f(x)=\exp(x)$ alors $f'(x)=\exp(x)$. On retrouve bien que $f'(0)=\exp(0)=1$.

1.2. Calcul approché de valeurs

Connaître la dérivée d'une fonction en un point permet d'approcher les valeurs de la fonction autour de ce point. Commençons par un petit changement de variable. Posons $h=x-x_0$, ce qui revient à écrire $x=x_0+h$ (et considérer un intervalle centré en x_0). Comme on s'intéresse aux valeurs de x qui tendent vers x_0, cela revient à dire que h tend vers 0. Le taux d'accroissement devient $\frac{f(x_0+h)-f(x_0)}{h}$ et on a :
$$\frac{f(x_0+h)-f(x_0)}{h}\xrightarrow[h\to 0]{}f'(x_0).$$
Cela fournit une valeur approchée de f en x_0+h, pourvu que h soit proche de 0 :
$$\boxed{f(x_0+h)\simeq f(x_0)+hf'(x_0).}$$

Démonstration. Comme
$$\frac{f(x_0+h)-f(x_0)}{h}\xrightarrow[h\to 0]{}f'(x_0),$$
alors pour h suffisamment petit :
$$\frac{f(x_0+h)-f(x_0)}{h}\simeq f'(x_0).$$
En multipliant de part et d'autre par h, on obtient l'estimation voulue. □

Exemple.
On souhaite trouver une valeur approchée de sin(0.01) sans calculatrice. Posons $f(x) = \sin(x)$. On sait que $f'(x) = \cos(x)$. Avec $x_0 = 0$, on a $f(x_0) = \sin(0) = 0$. On se doute bien que sin(0.01) sera proche de 0, mais on veut faire mieux. Posons $h = 0.01$ et calculons $f'(x_0) = \sin'(0) = \cos(0) = 1$. Donc notre formule s'écrit, pour h proche de 0 :

$$\sin(h) = f(x_0 + h) \simeq f(x_0) + hf'(x_0) = 0 + h \cdot 1 = h.$$

Et donc pour $h = 0.01$ on a $\sin(0.01) \simeq 0.01$ On vérifie à la calculatrice que $\sin(0.01) = 0.00999983\ldots$, donc notre approximation est très bonne. (Attention, il faut d'abord sélectionner les radians comme unité d'angle sur la calculatrice.)

Exemple.
Justifions la formule

$$\sqrt{1+h} \simeq 1 + \frac{1}{2}h,$$

valable pour des valeurs de h proches de 0.
Soit $f(x) = \sqrt{x}$ et $x_0 = 1$. On sait que $f'(x) = \frac{1}{2}\frac{1}{\sqrt{x}}$ donc $f'(x_0) = \frac{1}{2}$. Pour h proche de 0 :

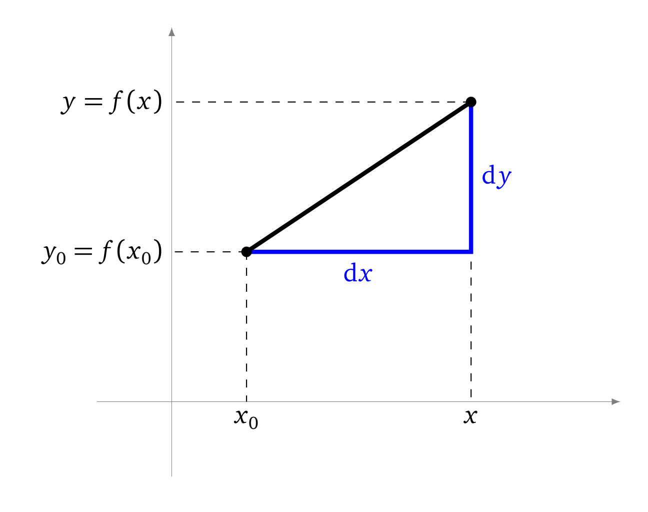
$$\sqrt{1+h} = f(x_0 + h) \simeq f(x_0) + hf'(x_0) = 1 + \frac{1}{2}h.$$

Par exemple avec $h = 0.1$ on obtient :

$$\sqrt{1.1} \simeq 1 + 0.5 \times 0.1 = 1.05$$

et la calculatrice donne $\sqrt{1.1} = 1.0488\ldots$

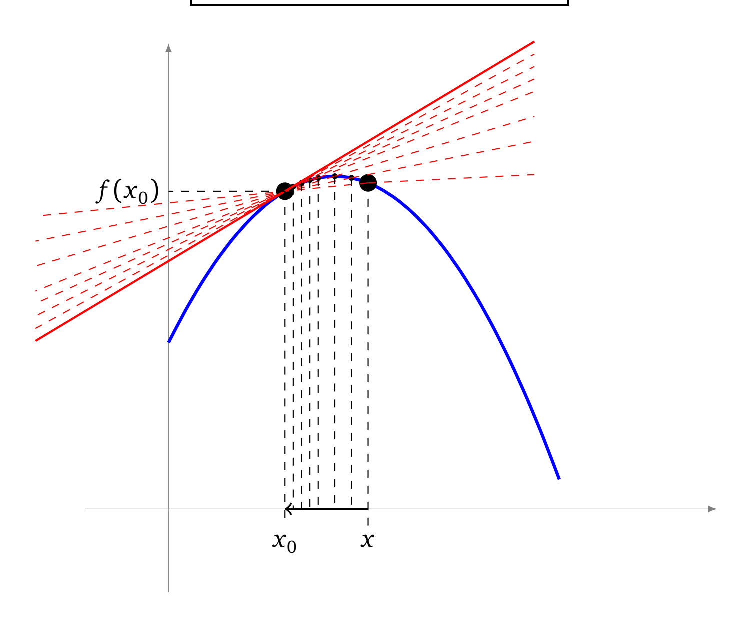
Remarque.
Voici quelques explications sur la notation $\frac{\mathrm{d}f}{\mathrm{d}x}$, préférée par les physiciens, et qu'il faut bien comprendre car nous allons la généraliser plus tard.
La notation « $\mathrm{d}x$ » représente un élément infinitésimal de la variable x, c'est-à-dire la valeur $x - x_0$, avec x très proche de x_0 (c'est-à-dire $x \to x_0$). $\mathrm{d}y$ ou encore $\mathrm{d}f$ représente la variation correspondante de la fonction, c'est-à-dire la valeur $f(x) - f(x_0)$, pour les mêmes valeurs de x et x_0. Ainsi $\frac{\mathrm{d}f}{\mathrm{d}x}(x_0)$ représente le quotient de ces deux valeurs, autrement dit le taux d'accroissement pris à la limite.



1.3. Tangente

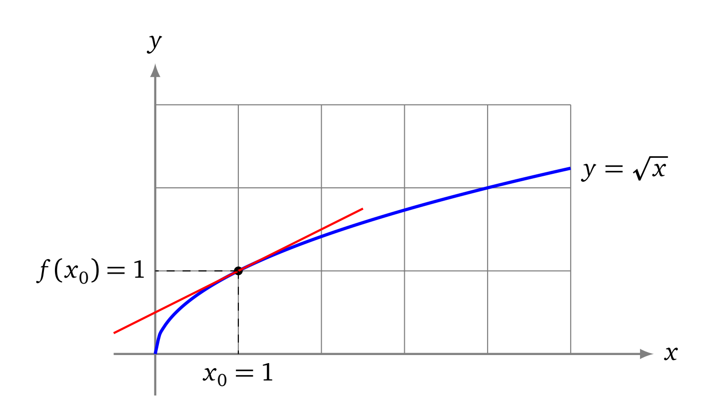
L'interprétation géométrique de la dérivée est essentielle ! Le coefficient directeur de la tangente au graphe de f en x_0 est $f'(x_0)$.
Une équation de la *tangente* au point $(x_0, f(x_0))$ est donc :

$$\boxed{y = (x - x_0)f'(x_0) + f(x_0)}$$



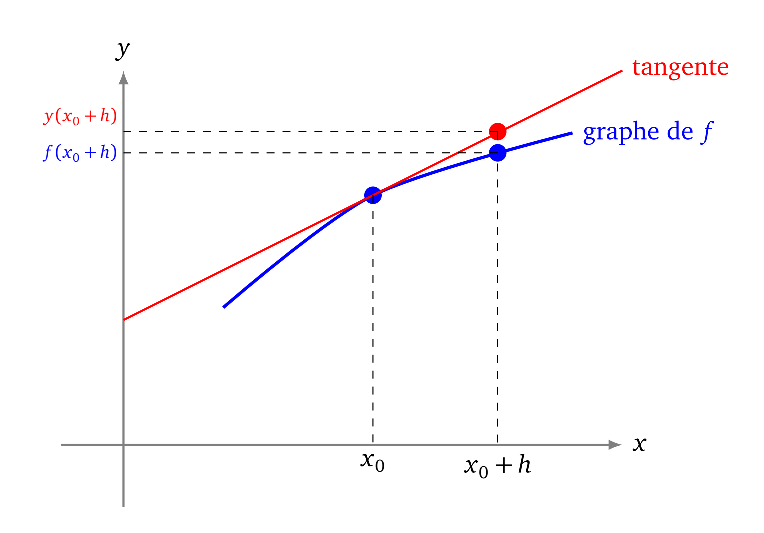
Justification : la droite qui passe par les points $(x_0, f(x_0))$ et $(x, f(x))$ a pour coefficient directeur $\frac{f(x)-f(x_0)}{x-x_0}$. À la limite, lorsque $x \to x_0$, on trouve que le coefficient directeur de la tangente est $f'(x_0)$. Voir l'illustration ci-dessus pour des valeurs de x tendant vers x_0.

Exemple.
Voici le graphe de la fonction définie par $f(x) = \sqrt{x}$. La dérivée est $f'(x) = \frac{1}{2}\frac{1}{\sqrt{x}}$. La tangente en $x_0 = 1$ a donc pour équation $y = (x-1)\frac{1}{2} + 1$, autrement dit c'est $y = \frac{1}{2}x + \frac{1}{2}$.



La tangente en x_0 est la droite qui « approche » au mieux le graphe de f autour x_0. Voici l'interprétation géométrique de l'approximation étudiée dans la section précédente : pour x proche de x_0, au lieu de lire les valeurs $f(x)$ sur le graphe de f, on lit les valeurs approchées $y(x) = (x - x_0)f'(x_0) + f(x_0)$ sur la tangente en x_0.





1.4. Formules usuelles

Voici les expressions des dérivées de fonctions classiques. Elles sont à connaître sur le bout des doigts.

Fonction	Dérivée
x^n	$nx^{n-1} \quad (n \in \mathbb{Z})$
$\frac{1}{x}$	$-\frac{1}{x^2}$
$\sqrt{x}$	$\frac{1}{2}\frac{1}{\sqrt{x}}$
x^α	$\alpha x^{\alpha-1} \quad (\alpha \in \mathbb{R})$
e^x	e^x
$\ln x$	$\frac{1}{x}$
$\cos x$	$-\sin x$
$\sin x$	$\cos x$
$\tan x$	$1+\tan^2 x = \frac{1}{\cos^2 x}$

Ces formules, conjuguées aux opérations décrites dans la proposition ci-dessous, permettent de calculer un très grand nombre de dérivées.

Proposition 1.
À partir de deux fonctions dérivables $f : I \to \mathbb{R}$ et $g : I \to \mathbb{R}$, on calcule la dérivée des opérations élémentaires suivantes :

- ***Somme***

$$\boxed{(f+g)' = f' + g'}$$

Autrement dit $(f+g)'(x) = f'(x) + g'(x)$ pour tout $x \in I$.

- ***Produit par un scalaire***

$$\boxed{(\lambda f)' = \lambda f'}$$

où λ est un réel fixé. Autrement dit $(\lambda f)'(x) = \lambda f'(x)$.

- ***Produit***

$$\boxed{(f \times g)' = f'g + fg'}$$

Autrement dit $(f \times g)'(x) = f'(x)g(x) + f(x)g'(x)$.

- ***Quotient***

$$\boxed{\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}}$$

Autrement dit $\left(\frac{f}{g}\right)'(x) = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$ *(si* $g(x) \neq 0$*).*

Exemple.

- Soit $f_1(x) = x^2 + \cos(x)$ alors $f_1'(x) = 2x - \sin(x)$. La dérivée d'une somme est la somme des dérivées.
- Soit $f_2(x) = x^3 \times \sin(x)$ alors $f_2'(x) = 3x^2 \sin(x) + x^3 \cos(x)$. Nous avons appliqué la formule de la dérivée d'un produit.
- Soit $f_3(x) = \frac{\ln(x)}{e^x}$ alors $f_3'(x) = \frac{\frac{1}{x}e^x - \ln(x)e^x}{e^{2x}}$ en appliquant la formule de la dérivée d'un quotient et la relation $(e^x)^2 = e^{2x}$.

Démonstration. Voyons comment prouver une des formules, par exemple $(f \times g)' = f'g + fg'$.
Fixons $x_0 \in I$. Nous allons réécrire le taux d'accroissement de $f(x) \times g(x)$:

$$\begin{aligned}\frac{f(x)g(x) - f(x_0)g(x_0)}{x - x_0} &= \frac{f(x) - f(x_0)}{x - x_0}g(x) + \frac{g(x) - g(x_0)}{x - x_0}f(x_0) \\ &\xrightarrow[x \to x_0]{} f'(x_0)g(x_0) + g'(x_0)f(x_0).\end{aligned}$$

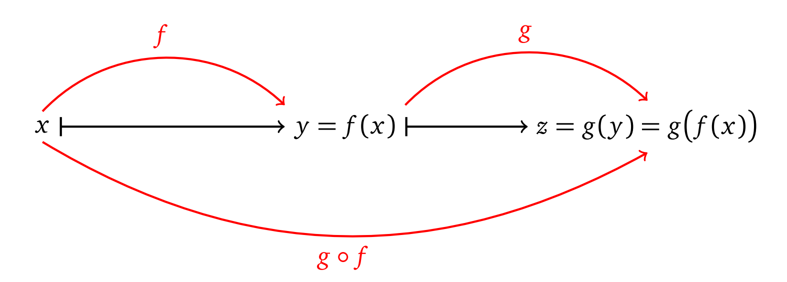
Ceci étant vrai pour tout $x_0 \in I$, la fonction $f \times g$ est dérivable sur I et a pour dérivée $f'g + fg'$. □

1.5. Dérivée d'une composition

Soient $f : \mathbb{R} \to \mathbb{R}$ et $g : \mathbb{R} \to \mathbb{R}$ deux fonctions. La ***composition*** $g \circ f$ est définie par :

$$g \circ f(x) = g\big(f(x)\big).$$

Il faut bien faire attention à l'ordre : on calcule d'abord $f(x)$ puis on évalue la quantité $g\big(f(x)\big)$ à l'aide de la fonction g.



La dérivée d'une composition est la formule fondamentale de tout ce cours ! En effet, c'est cette formule qui permet de calculer les bons paramètres pour un réseau de neurones.
Voici la formule :

Proposition 2.
La dérivée de $g \circ f$ *est donnée par :*

$$\boxed{\big(g \circ f\big)'(x) = g'\big(f(x)\big) \cdot f'(x)}$$

Autrement dit, si $F(x) = g\big(f(x)\big)$ alors $F'(x) = g'\big(f(x)\big) \cdot f'(x)$.
Voici une façon pratique et mnémotechnique de retenir la formule : on note $y = f(x)$ et $z = g(y) = g \circ f(x)$.
Alors

$$\boxed{\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \times \frac{\mathrm{d}y}{\mathrm{d}x}}$$

qui est exactement notre formule :

$$\frac{\mathrm{d}g \circ f}{\mathrm{d}x}(x) = \frac{\mathrm{d}g}{\mathrm{d}y}(y) \times \frac{\mathrm{d}f}{\mathrm{d}x}(x).$$

Pourquoi cette formule est-elle si facile à retenir ? C'est comme si les dérivées se comportaient comme des fractions que l'on pouvait simplifier. Bien sûr c'est juste la notation qui permet cela.

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\cancel{\mathrm{d}y}} \times \frac{\cancel{\mathrm{d}y}}{\mathrm{d}x}$$

Démonstration. La preuve est similaire à celle du produit de deux fonctions, si on suppose que $f(x) \neq f(x_0)$ pour $x \neq x_0$ (autour de x_0), en écrivant cette fois :

$$\begin{aligned}\frac{g \circ f(x) - g \circ f(x_0)}{x - x_0} &= \frac{g\big(f(x)\big) - g\big(f(x_0)\big)}{f(x) - f(x_0)} \times \frac{f(x) - f(x_0)}{x - x_0} \\ &\xrightarrow[x \to x_0]{} g'\big(f(x_0)\big) \times f'(x_0).\end{aligned}$$

□

Exemple.
Commençons par un exemple simple avec $F(x) = \sin(3x)$. C'est la composition de $f(x) = 3x$ avec $g(x) = \sin(x)$: $F(x) = g \circ f(x)$. On sait que $f'(x) = 3$ et $g'(x) = \cos(x)$ donc $g'(f(x)) = \cos(3x)$. Ainsi

$$F'(x) = 3\cos(3x).$$

Une autre façon de le voir est de poser $y = 3x$, $z = \sin(y)$ $(= \sin(3x) = F(x))$. On a $\frac{\mathrm{d}z}{\mathrm{d}y} = \cos(y)$ et $\frac{\mathrm{d}y}{\mathrm{d}x} = 3$, ainsi :

$$F'(x) = \frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \times \frac{\mathrm{d}y}{\mathrm{d}x} = \cos(y) \times 3.$$

Et comme $y = 3x$, on obtient :

$$F'(x) = 3\cos(3x).$$

Exemple.
Soit $F(x) = \ln(\cos(x))$. C'est la composition de $f(x) = \cos(x)$ avec $g(x) = \ln(x)$: $F(x) = g \circ f(x)$. On sait que $f'(x) = -\sin(x)$. D'autre part $g'(x) = \frac{1}{x}$, donc $g'(f(x)) = \frac{1}{f(x)} = \frac{1}{\cos(x)}$. Ainsi :

$$F'(x) = \frac{-\sin(x)}{\cos(x)}.$$

Exemple.
Soit $F(x) = \sqrt{\tan(x)}$. Cette fois utilisons la notation des physiciens avec $y = \tan(x)$ et $z = \sqrt{y}$ $(= F(x))$. Alors :

$$F'(x) = \frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \times \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{1}{2}\frac{1}{\sqrt{y}} \times (1 + \tan^2(x)).$$

Et comme $y = \tan(x)$, on obtient :

$$F'(x) = \frac{1}{2}\frac{1 + \tan^2(x)}{\sqrt{\tan(x)}}.$$

Voici ce que donnent les dérivées des compositions des fonctions usuelles. Les formules s'obtiennent en appliquant directement la formule donnée plus haut, mais sont tout de même à connaître par cœur. Ici u désigne une fonction $x \mapsto u(x)$.

Fonction	**Dérivée**
u^n	$nu'u^{n-1} \quad (n \in \mathbb{Z})$
$\frac{1}{u}$	$-\frac{u'}{u^2}$
$\sqrt{u}$	$\frac{1}{2}\frac{u'}{\sqrt{u}}$
u^α	$\alpha u'u^{\alpha-1} \quad (\alpha \in \mathbb{R})$
e^u	$u'e^u$
$\ln u$	$\frac{u'}{u}$
$\cos u$	$-u' \sin u$
$\sin u$	$u' \cos u$
$\tan u$	$u'(1+\tan^2 u) = \frac{u'}{\cos^2 u}$

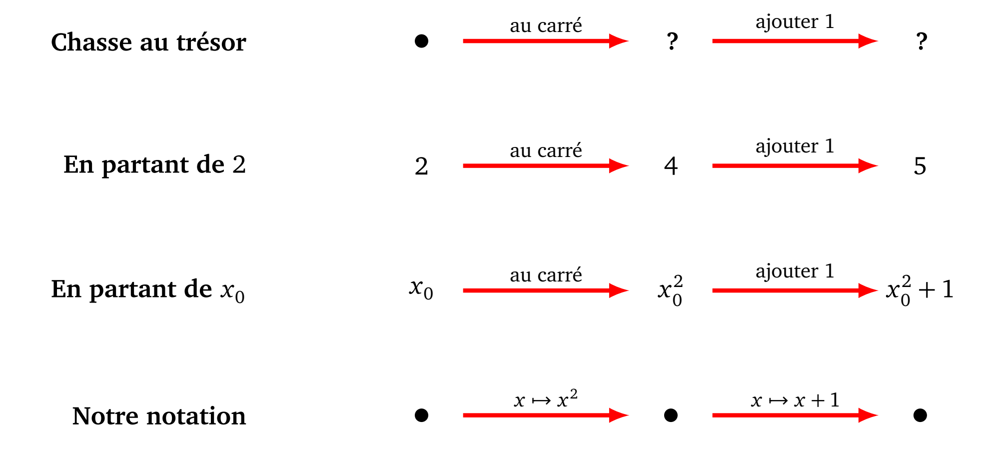
La formule de la dérivée de composition se généralise : si x est une fonction de t, y est une fonction de x et z est une fonction de y alors :

$$\frac{dz}{dt} = \frac{dz}{dy} \times \frac{dy}{dx} \times \frac{dx}{dt}.$$

2. Dérivation automatique

2.1. Graphe de calcul

Un graphe de calcul est comme une chasse au trésor : on effectue des calculs étape par étape jusqu'à obtenir le résultat final. Voici un exemple ci-dessous.



On part d'un nombre et on doit calculer des résultats en fonction des opérations précisées sur les flèches (ici mettre le nombre au carré, puis ajouter 1). Si on part par exemple de 2 : on l'élève au carré, puis on ajoute 1 pour obtenir le résultat 5. Si on partait d'un réel quelconque x_0, cela donne $x_0^2 + 1$. Le dernier diagramme est le graphe de calcul tel que nous allons le représenter dans la suite.

Plus généralement, on représente l'évaluation par une fonction f comme ceci :

$$\bullet \xrightarrow{x \mapsto f(x)} \bullet$$

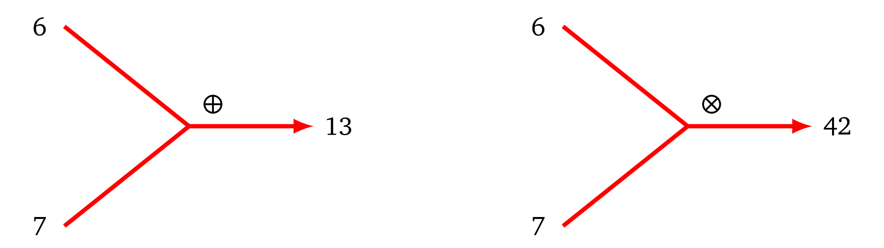
ce qui permet de représenter facilement la composition de deux fonctions $g \circ f$:

$$\bullet \xrightarrow{x \mapsto f(x)} \bullet \xrightarrow{x \mapsto g(x)} \bullet$$

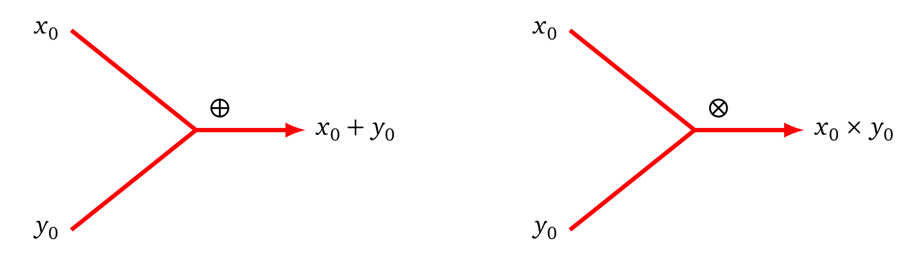
qui correspond au calcul :

$$x_0 \xrightarrow{x \mapsto f(x)} f(x_0) \xrightarrow{x \mapsto g(x)} g\big(f(x_0)\big)$$

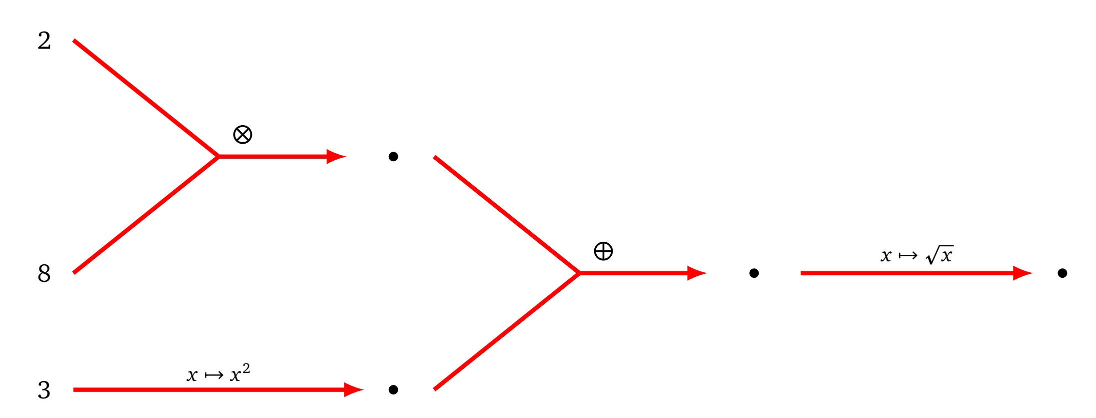
Pour les additions et les multiplications, chaque graphe possède deux entrées et une sortie.
Par exemple :



Et plus généralement :



Voici un graphe de calcul obtenu en combinant plusieurs opérations. Vérifier que le résultat final (tout à droite) vaut 5. Écrire le calcul algébrique effectué en remplaçant les nombres 2, 8 et 3 par trois variables x_0, y_0 et z_0.



2.2. Dérivation automatique (cas simple)

Nous allons enrichir nos graphes de calcul à l'aide de la dérivée : en plus d'écrire le résultat du calcul $f(x_0)$, on ajoute entre crochets la valeur de la dérivée $f'(x_0)$. On appelle cette valeur la ***dérivée locale***. Pour aider à effectuer les calculs, on peut rappeler sous la flèche la formule de la fonction dérivée.

$$x_0 \xrightarrow[{[x \mapsto f'(x)]}]{x \mapsto f(x)} \begin{matrix} f(x_0) \\ [f'(x_0)] \end{matrix}$$

Voici trois exemples :

$$2 \xrightarrow[{[x\mapsto \frac{1}{x}]}]{x\mapsto \ln(x)} \ln(2)\ [\tfrac{1}{2}] \qquad -1 \xrightarrow[{[x\mapsto -2e^{-2x}]}]{x\mapsto e^{-2x}} e^2\ [-2e^2] \qquad 3 \xrightarrow{x\mapsto x^2} 9\ [6]$$

Bien sûr, il n'est pas obligatoire d'écrire la formule $[x \mapsto f'(x)]$ sous la flèche quand on connaît bien ses dérivées !
Voici comment évaluer la dérivée d'une composition. Prenons la fonction $F(x) = \cos(2x^2)$ pour laquelle on souhaite par exemple calculer $F'(3)$.
Une première méthode serait de calculer $F'(x)$ (quel que soit x), puis évaluer $F'(3)$. Voici une autre méthode : on pose $f(x) = 2x^2$ et $g(y) = \cos(y)$ alors $F(x) = g \circ f(x)$ (on aurait pu noter $g(x) = \cos(x)$ mais utiliser le nom y est plus pratique pour la suite).

$$3 \xrightarrow{x\mapsto 2x^2} \bullet \xrightarrow{y\mapsto \cos(y)} \bullet$$

Première étape : graphe de calcul. On écrit le graphe de calcul correspondant à cette composition :

$$3 \xrightarrow{x\mapsto 2x^2} 18 \xrightarrow{y\mapsto \cos(y)} \cos(18)$$

Seconde étape : ajout des dérivées locales. On calcule les valeurs de dérivées locales :

$$3 \xrightarrow[{[x\mapsto 4x]}]{x\mapsto 2x^2} 18\ [12] \xrightarrow[{[y\mapsto -\sin(y)]}]{y\mapsto \cos(y)} \cos(18)\ [-\sin(18)]$$

Troisième étape : calcul de la dérivée de la composition. On calcule la dérivée de la composition comme le produit des dérivées locales.
On note $\{\{F'(3)\}\}$ cette dérivée, entre accolades doubles.

$$x=3 \xrightarrow[{[x\mapsto 4x]}]{x\mapsto 2x^2} y=18\ [12] \xrightarrow[{[y\mapsto -\sin(y)]}]{y\mapsto \cos(y)} z=\cos(18)\ [-\sin(18)]$$

$\times$

$\{\{-12\sin(18)\}\}$

Pour des raisons que l'on expliquera plus tard, il faut prendre l'habitude de calculer le produit de la droite vers la gauche : on effectue $(-\sin(18)) \times 12$ et non $12 \times (-\sin(18))$.
Reprenons la même fonction, mais cette fois pour calculer $F'(4)$:

Exemple avec $x = 4$

$$4 \xrightarrow{x\mapsto 2x^2} \bullet \xrightarrow{y\mapsto \cos(y)} \bullet$$

Graphe de calcul

$$4 \xrightarrow{x\mapsto 2x^2} 32 \xrightarrow{y\mapsto \cos(y)} \cos(32)$$

Dérivées locales

$$4 \xrightarrow[{[x\mapsto 4x]}]{x\mapsto 2x^2} 32\ [16] \xrightarrow[{[y\mapsto -\sin(y)]}]{y\mapsto \cos(y)} \cos(32)\ [-\sin(32)]$$

Dérivée en $x = 4$

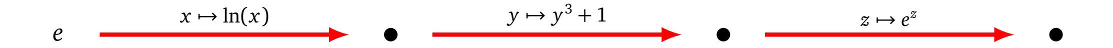
$$x=4 \xrightarrow[{[x\mapsto 4x]}]{x\mapsto 2x^2} y=32\ [16] \xrightarrow[{[y\mapsto -\sin(y)]}]{y\mapsto \cos(y)} z=\cos(32)\ [-\sin(32)]$$

$\times$

$\{\{-16\sin(32)\}\}$

On trouve $F'(4) = -16\sin(32)$.

Plus de fonctions. Le principe est le même sur des exemples plus compliqués, ici avec trois fonctions : $F = h \circ g \circ f$.



Qui donne :

e $\xrightarrow[{[x \mapsto 1/x]}]{x \mapsto \ln(x)}$ 1 $[1/e]$ $\xrightarrow[{[y \mapsto 3y^2]}]{y \mapsto y^3+1}$ 2 $[3]$ $\xrightarrow[{[z \mapsto e^z]}]{z \mapsto e^z}$ e^2 $[e^2]$

$\{\{e^2 \times 3 \times \frac{1}{e} = 3e\}\}$

Répondre aux questions suivantes :

- Quelle est la fonction composée ?
- Quelle est la dérivée calculée ? Combien vaut-elle ? Vérifier les calculs !
- Combien vaudrait la dérivée en $x = 1$? En $x = 2$?

Justification. Pourquoi ces opérations donnent-elles le bon résultat ? Il s'agit juste de la réécriture de la formule de la dérivée d'une composition !
Pour $F = g \circ f$, les dérivées locales sont $f'(x)$ et $g'(y)$ (en notant $y = f(x)$), le produit des dérivées locales est donc $f'(x) \times g'(y) = f'(x) \times g'(f(x)) = F'(x)$. C'est encore plus facile à comprendre en notant $y = f(x)$, $z = g(y) = g \circ f(x)$: il s'agit juste de la formule :

$$\frac{\mathrm{d}z}{\mathrm{d}y} \times \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}x}.$$

x $\xrightarrow{x \mapsto f(x)}$ $y = f(x)$ $[f'(x) = \frac{dy}{dx}]$ $\xrightarrow{y \mapsto g(y)}$ $z = g(y) = g \circ f(x)$ $[g'(y) = \frac{dz}{dy}]$

$\times$

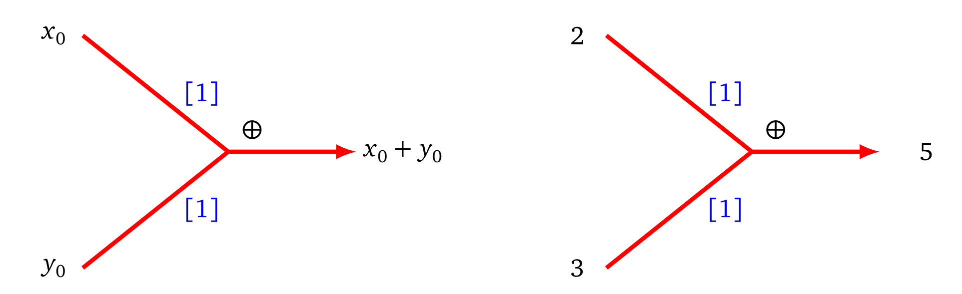
$\{\{(g \circ f)'(x) = \frac{dz}{dx} = \frac{dz}{dy} \times \frac{dy}{dx}\}\}$

Pour des compositions plus compliquées, par exemple avec trois compositions, c'est la réécriture de la formule :

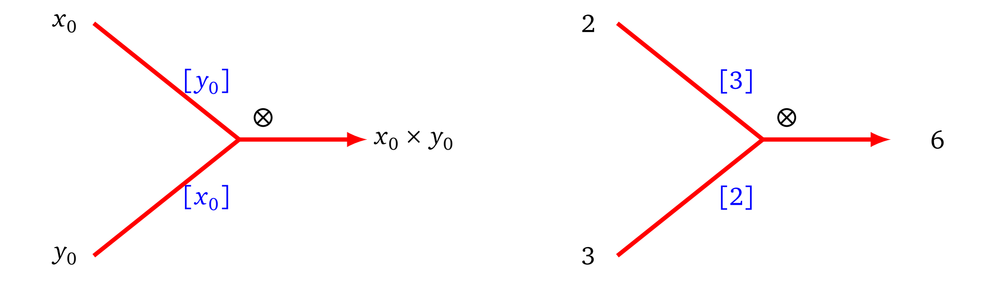
$$\frac{\mathrm{d}F}{\mathrm{d}x} = \frac{\mathrm{d}F}{\mathrm{d}z} \times \frac{\mathrm{d}z}{\mathrm{d}y} \times \frac{\mathrm{d}y}{\mathrm{d}x}.$$

2.3. Dérivation automatique (cas général)

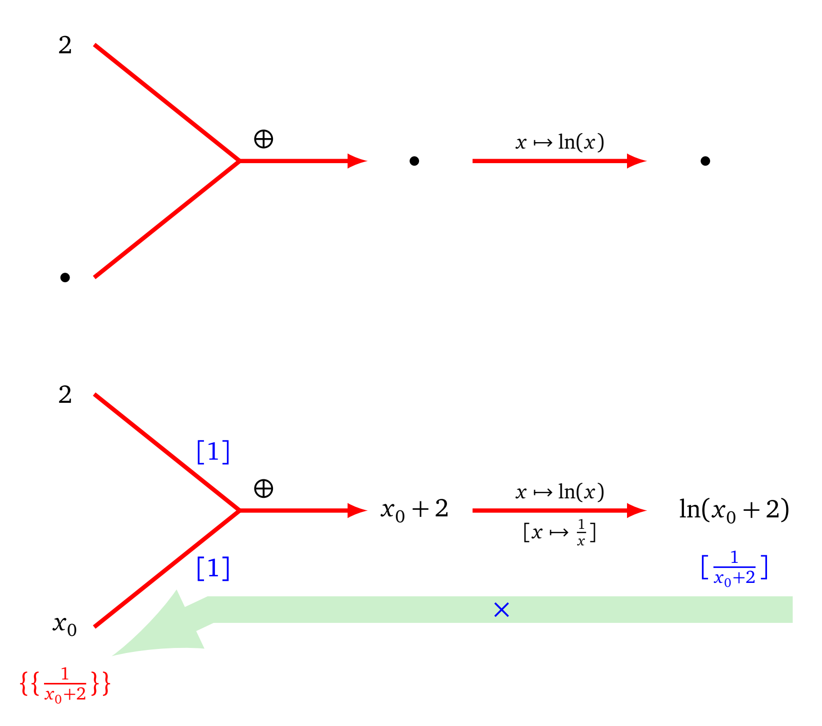
Pour les additions et multiplications, on ajoute des dérivées locales sur les arêtes. Les formules seront justifiées dans le chapitre « Gradient ». Pour l'addition, les deux dérivées locales valent 1 (le principe à gauche ci-dessous, un exemple à droite).



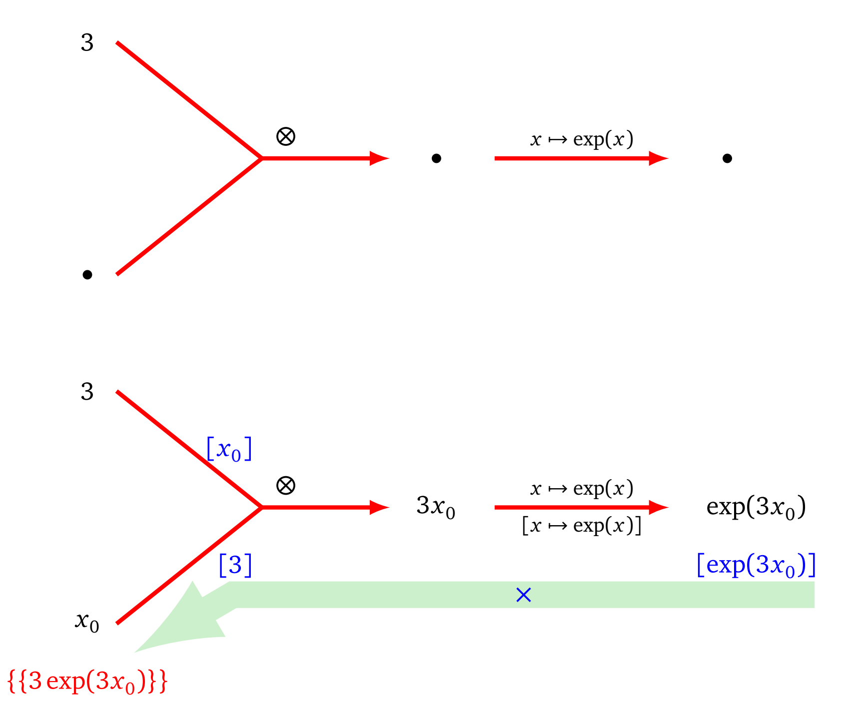
Pour la multiplication, les dérivées locales correspondent aux valeurs d'entrée x_0 et y_0, mais échangées : $[y_0]$ et $[x_0]$ (le principe à gauche ci-dessous, un exemple à droite).



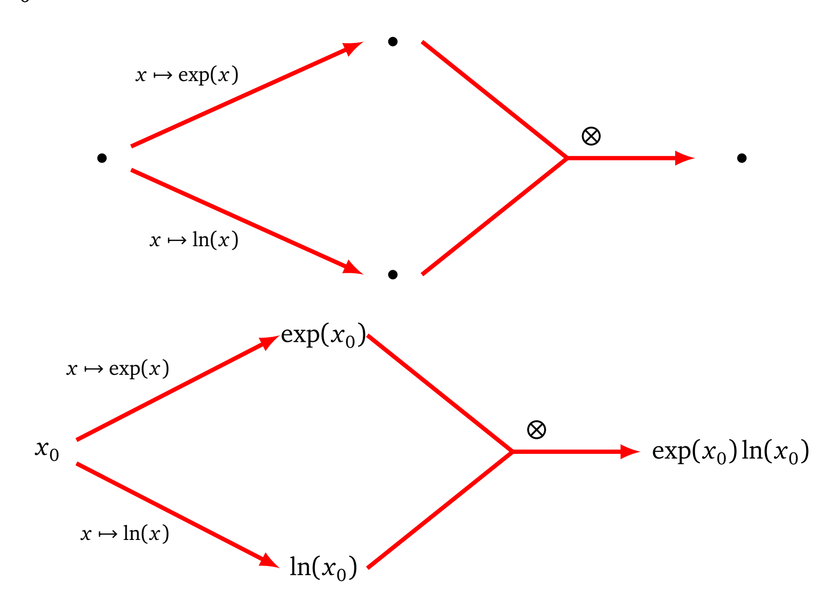
Pour calculer la dérivée d'une composition, le principe est le même qu'auparavant : la dérivée est le produit des dérivées locales le long des arêtes joignant la valeur x_0 à la valeur $F(x_0)$.
Par exemple, voici le graphe de calcul correspondant à $F(x) = \ln(x+2)$, on trouve $F'(x) = \frac{1}{x+2}$.



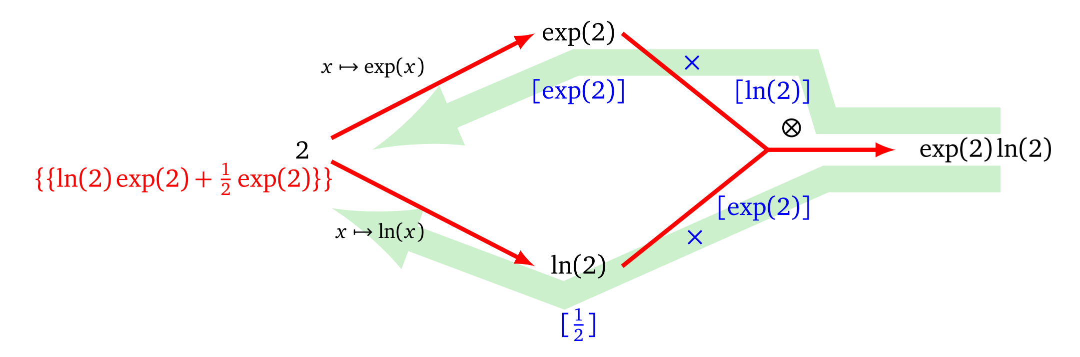
Voici le graphe de calcul correspondant à $F(x) = \exp(3 \times x)$, on trouve $F'(x) = 3\exp(3x)$.



Il existe une nouvelle situation, lorsque l'on a besoin de dupliquer la variable d'entrée. Considérons par exemple la fonction $F(x) = \exp(x) \times \ln(x)$. L'entrée « • » tout à gauche (sur la figure ci-dessous) représente une même valeur x_0 utilisée deux fois.



Voici le graphe de calcul correspondant à $F(2)$ pour l'entrée $x_0 = 2$, complété par les dérivées locales (entre crochets) et la dérivée globale par rapport à la variable d'entrée (entre accolades doubles) :



La dérivée $F'(2)$ est la somme de termes, calculée entre accolades :

$$F'(2) = \{\{\ln(2)\exp(2) + \frac{1}{2}\exp(2)\}\} = \left(\ln(2) + \frac{1}{2}\right)e^2.$$

Bilan.

> Pour calculer la dérivée d'une fonction composée F en x_0 :
> - pour chaque entrée, calculer le produit des dérivées locales (qui sont entre crochets) le long des chemins allant de la valeur finale $F(x_0)$ en revenant vers l'entrée,
> - puis, calculer la somme de ces produits (écrite entre accolades doubles).

2.4. Dérivées séquentielles

Cette section est plus technique et peut être omise en première lecture.

Soit $F(x) = h \circ g \circ f(x)$ une composition de trois fonctions. Notons

$$y = f(x), \qquad z = g(y), \qquad t = h(z).$$

On appelle ***dérivées séquentielles***, les dérivées de F par rapport à chacune des variables (ici x, y, z, t).

$$x \xrightarrow{f} y = f(x)\ [y' = \tfrac{df}{dx}] \xrightarrow{g} z = g(y)\ [z' = \tfrac{dg}{dy}] \xrightarrow{h} t = h(z)\ [t' = \tfrac{dh}{dz}]$$

$$\{\{\tfrac{dF}{dx}\}\} \qquad \{\{\tfrac{dF}{dy}\}\} \qquad \{\{\tfrac{dF}{dz}\}\} \qquad \{\{\tfrac{dF}{dt} = 1\}\}$$

Dans ce graphe de calcul, les dérivées locales sont entre crochets. Les dérivées séquentielles sont entre accolades doubles (lues en partant de la droite à partir de la figure ci-dessus) :

$$\frac{\mathrm{d}F}{\mathrm{d}t} = \frac{\mathrm{d}t}{\mathrm{d}t} = 1, \qquad \frac{\mathrm{d}F}{\mathrm{d}z} = \frac{\mathrm{d}t}{\mathrm{d}z} = [t'],$$

$$\frac{\mathrm{d}F}{\mathrm{d}y} = \frac{\mathrm{d}t}{\mathrm{d}y} = \frac{\mathrm{d}t}{\mathrm{d}z} \times \frac{\mathrm{d}z}{\mathrm{d}y} = [t'] \times [z'], \qquad \frac{\mathrm{d}F}{\mathrm{d}x} = \frac{\mathrm{d}t}{\mathrm{d}x} = \frac{\mathrm{d}t}{\mathrm{d}z} \times \frac{\mathrm{d}z}{\mathrm{d}y} \times \frac{\mathrm{d}y}{\mathrm{d}x} = [t'] \times [z'] \times [y'].$$

On calcule donc facilement la dérivée $\frac{\mathrm{d}F}{\mathrm{d}x}$ mais on a aussi comme étapes intermédiaires $\frac{\mathrm{d}F}{\mathrm{d}y}$ et $\frac{\mathrm{d}F}{\mathrm{d}z}$.

Pourquoi est-il pertinent de calculer les dérivées séquentielles de la droite vers la gauche ? Sur l'exemple ci-dessus les dérivées séquentielles sont :

$$\{\{1\}\}, \qquad \{\{t'\}\}, \qquad \{\{t' \times z'\}\}, \qquad \{\{t' \times z' \times y'\}\}.$$

Pour éviter de calculer beaucoup de produits, on remarque que chaque nouveau produit est la multiplication de la dérivée séquentielle précédente avec la dérivée locale :

$$\{\{1\}\}, \qquad \{\{t'\}\} = \{\{1\}\} \times [t'], \qquad \{\{t' \times z'\}\} = \{\{t'\}\} \times [z'], \qquad \{\{t' \times z' \times y'\}\} = \{\{t' \times z'\}\} \times [y'].$$

Plus généralement, si $F = f_n \circ \cdots \circ f_2 \circ f_1$ et $x_1 = f_1(x_0)$, $x_2 = f_2(x_1)$,..., $x_n = f_n(x_{n-1})$ alors :

$$\left\{\left\{\frac{\mathrm{d}F}{\mathrm{d}x_i}\right\}\right\} = \left\{\left\{\frac{\mathrm{d}F}{\mathrm{d}x_{i+1}}\right\}\right\} \times \left[\frac{\mathrm{d}f_{i+1}}{\mathrm{d}x_i}\right].$$

$$x_i \xrightarrow{f_{i+1}} x_{i+1} = f_{i+1}(x_i)\ [x'_{i+1} = \tfrac{df_{i+1}}{dx_i}] \xrightarrow{f_{i+2}} x_{i+2}$$

$$\{\{\tfrac{dF}{dx_i}\}\} \qquad \{\{\tfrac{dF}{dx_{i+1}}\}\}$$

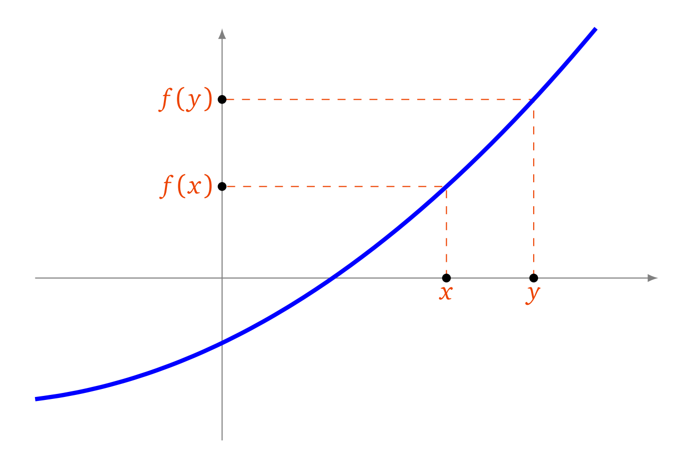
Pour obtenir toutes les dérivées séquentielles $\{\{\frac{\mathrm{d}F}{\mathrm{d}x_i}\}\}$, il y a donc n produits à calculer en partant de $\{\{\frac{\mathrm{d}F}{\mathrm{d}x_n}\}\}$ et en terminant par $\{\{\frac{\mathrm{d}F}{\mathrm{d}x_0}\}\}$. Si on n'utilise pas cette formule de récurrence alors on a $\frac{n(n+1)}{2}$ produits à calculer, ce qui est beaucoup plus si n est grand.

3. Étude de fonctions

3.1. Variations

Soit une fonction $f : I \to \mathbb{R}$.

Définition.
Pour tout x et y de I :
- f est ***croissante*** si $x \leqslant y$ implique $f(x) \leqslant f(y)$.
- f est ***strictement croissante*** si $x < y$ implique $f(x) < f(y)$.
- f est ***décroissante*** si $x \leqslant y$ implique $f(x) \geqslant f(y)$.
- f est ***strictement décroissante*** si $x < y$ implique $f(x) > f(y)$.



La croissance et la dérivée sont liées par la proposition suivante.

Proposition 3.
Soit $f : I \to \mathbb{R}$ une fonction dérivable, définie sur un intervalle I. Alors :
- *f est croissante sur $I \iff f'(x) \geqslant 0$ pour tout $x \in I$,*
- *f est décroissante sur $I \iff f'(x) \leqslant 0$ pour tout $x \in I$,*
- *f est constante sur $I \iff f'(x) = 0$ pour tout $x \in I$.*

On peut améliorer un peu certains résultats : par exemple si $f'(x) > 0$ pour tout $x \in I$ alors f est strictement croissante (la réciproque est fausse comme le montre l'exemple défini par $f(x) = x^3$).

Démonstration. Nous allons seulement prouver l'implication : f est croissante $\implies f'(x) \geqslant 0$. Nous admettrons les autres propriétés.
Soit f une fonction croissante. Fixons x_0 et soit $x \geqslant x_0$. Alors comme f est croissante on a $f(x) \geqslant f(x_0)$. Ainsi le taux d'accroissement $\frac{f(x)-f(x_0)}{x-x_0}$ est positif, pour tout $x > x_0$. Ainsi à la limite (lorsque $x \to x_0$ en ayant $x > x_0$) on a : $f'(x_0) = \lim_{x \to x_0^+} \frac{f(x)-f(x_0)}{x-x_0} \geqslant 0$.
□

Définition.
Une fonction $f : I \to J$ est une ***bijection*** si chaque $y \in J$ admet un unique antécédent $x \in I$, c'est-à-dire tel que $f(x) = y$.

Proposition 4.
Soit $f : I \to \mathbb{R}$ une fonction. Soit $J = f(I)$ l'ensemble des valeurs prises par f sur I. Si f est strictement croissante sur I (ou bien strictement décroissante sur I) alors la fonction $f : I \to J$ est bijective.

Démonstration. Par définition de J, tout $y \in J$ admet un antécédent $x \in I$ par f. Comme f est strictement croissante alors cet antécédent est unique. □

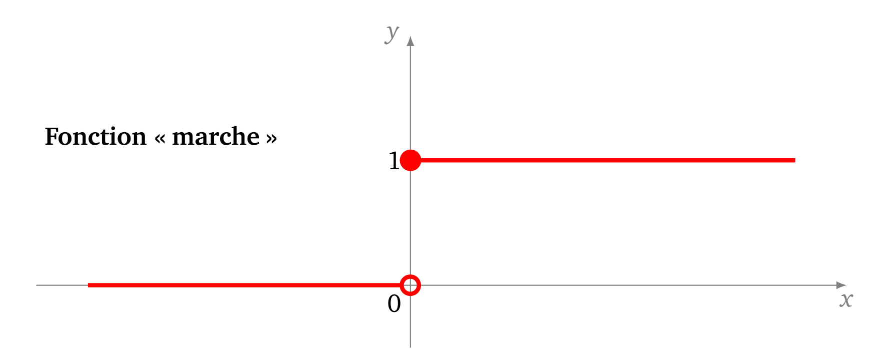
3.2. Fonctions d'activation

Une *fonction d'activation* n'est rien d'autre qu'une fonction $f : \mathbb{R} \to \mathbb{R}$ qui sera dans la suite associée à un neurone. Nous allons en étudier plusieurs exemples. À une *entrée* $x \in \mathbb{R}$, on associe une *sortie* $y = f(x)$.

Fonction marche de Heaviside.

C'est la fonction *marche d'escalier* définie par la formule suivante :

$$\begin{cases} H(x) = 0 & \text{si } x < 0 \\ H(x) = 1 & \text{si } x \geqslant 0 \end{cases}$$



Cela justifie le nom de fonction d'activation. Imaginons un composant électronique qui allume une diode lorsque l'intensité est positive. On modélise la situation ainsi :

- $x \in \mathbb{R}$ est l'intensité donnée en entrée,
- $y = 0$ (diode éteinte) ou $y = 1$ (diode allumée) est la valeur de sortie,
- la relation entre les deux est donnée par la fonction de Heaviside $y = H(x)$: la diode est activée lorsque l'intensité x est positive.

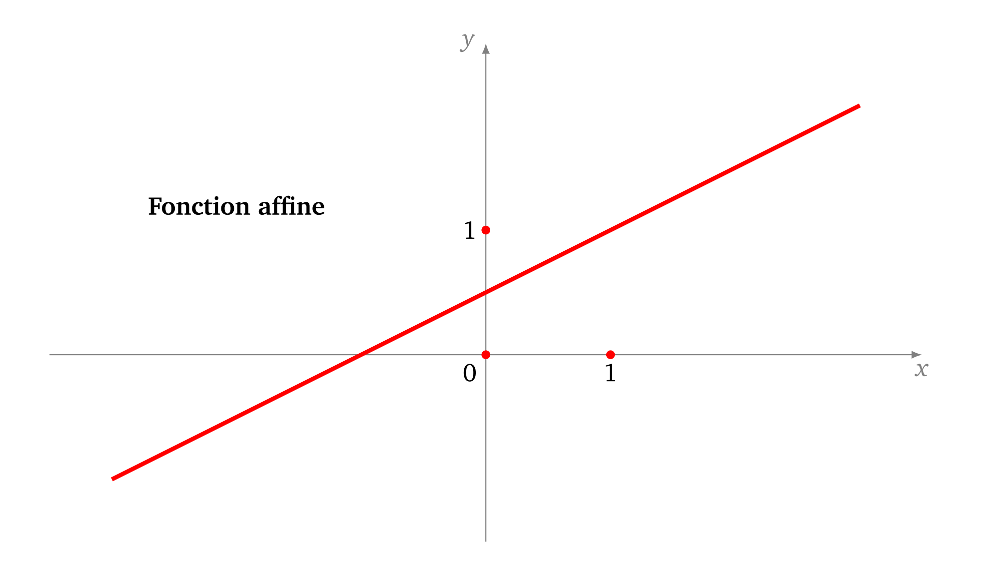
Remarque.
Quelques caractéristiques de H :

- la fonction a l'avantage d'être très simple et de séparer clairement les résultats en deux catégories (0 et 1),
- c'est aussi un inconvénient car il n'y a pas de nuance possible (du genre « plutôt oui/plutôt non »),
- la fonction H n'est pas dérivable en 0 (ni même continue), ce qui sera gênant pour la suite,
- avoir posé $H(0) = 1$ est un choix arbitraire, on aurait pu choisir une autre valeur comme 0 (certains préfèrent la valeur $\frac{1}{2}$).

Fonction affine.

Une *fonction affine* est définie par :

$$f(x) = ax + b$$



Remarque.

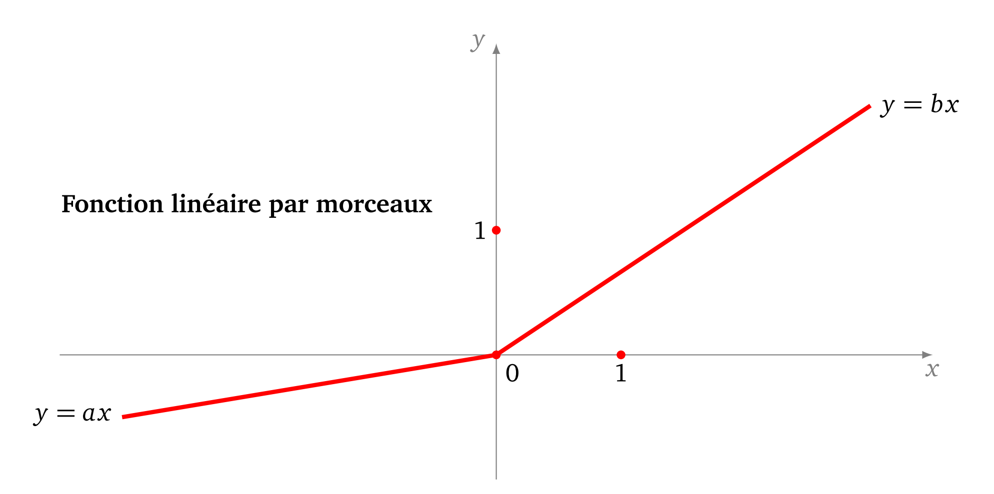
- Les fonctions affines seront à la base des réseaux de neurones. Elles sont simples.
- Si $a = 1$ et $b = 0$ alors $f(x) = x$, c'est la fonction identité : la sortie est égale à l'entrée. Nous l'utiliserons plus tard dans certains cas.
- Si $b = 0$ la sortie est proportionnelle à l'entrée.
- Le principal inconvénient est le suivant : si $f(x) = ax + b$ et $g(x) = cx + d$ sont deux fonctions affines alors la composition $h = g \circ f$ est encore une fonction affine. Ainsi, avec des fonctions d'activation qui sont affines, par composition on n'obtiendra que des fonctions affines, ce qui n'est pas assez riche pour les problèmes de classification.

ReLU et fonctions linéaires par morceaux.

Une fonction définie par

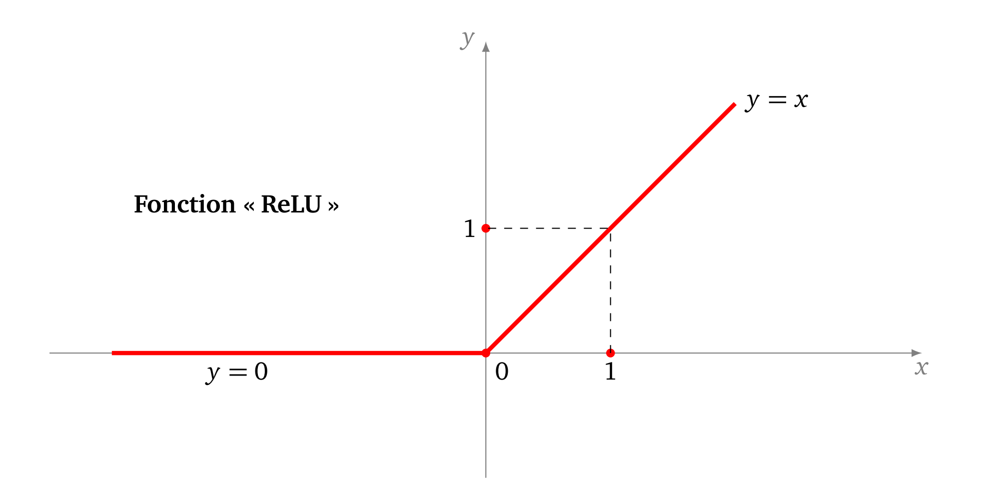
$$\begin{cases} f(x) = ax & \text{si } x < 0 \\ f(x) = bx & \text{si } x \geqslant 0 \end{cases}$$

est un exemple de ***fonction linéaire par morceaux*** (elle est linéaire à gauche et linéaire à droite).



La plus utilisée est la fonction ReLU (pour *Rectified Linear Unit*), définie par

$$\begin{cases} f(x) = 0 & \text{si } x < 0 \\ f(x) = x & \text{si } x \geqslant 0 \end{cases}$$



Remarque.

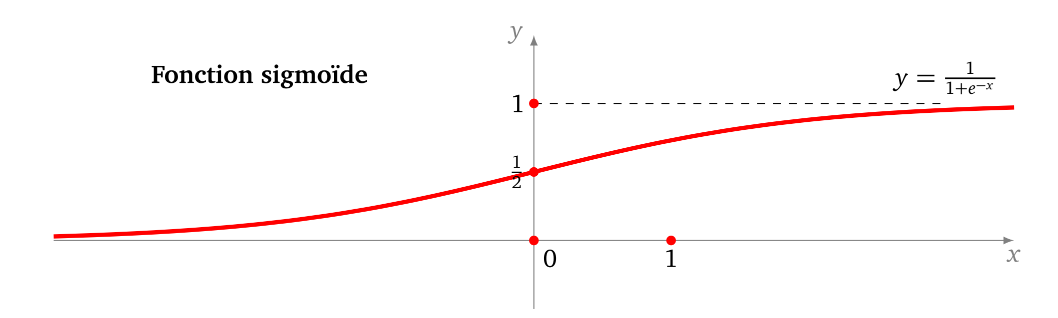
- Attention ces fonctions ne sont pas dérivables en 0 (sauf dans le cas $a = b$).
- La fonction ReLU conjugue les bénéfices d'une fonction continue, avec une activation (ici la sortie est non nulle pour une entrée strictement positive) et d'une sortie proportionnelle à l'entrée (pour les entrées positives).

La fonction sigmoïde.

La *fonction sigmoïde* est définie par :

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Le graphe de la fonction σ possède une forme de « sigma » comme le symbole intégrale $\int$.



La fonction ressemble un peu à la fonction de Heaviside mais elle à l'avantage d'être continue et dérivable. Elle propose une transition douce de 0 à 1, la transition étant assez linéaire autour de 0.
Étudions en détails cette fonction.

Proposition 5.

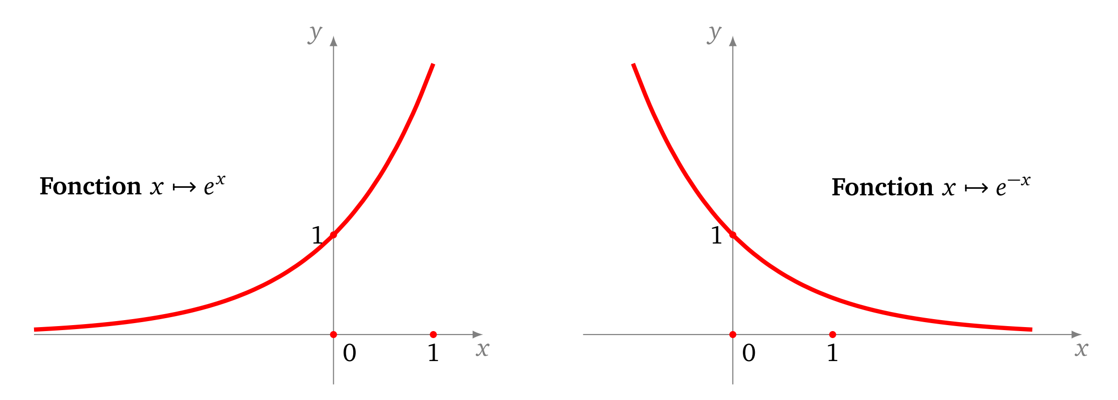
1. *La fonction σ est strictement croissante.*
2. *La limite en $-\infty$ est 0, la limite en $+\infty$ en 1.*
3. *La fonction σ est continue et dérivable et*

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}.$$

4. *La tangente en $x = 0$ a pour équation $y = \frac{1}{4}x + \frac{1}{2}$.*
5. *La fonction dérivée vérifie aussi la relation :*

$$\boxed{\sigma'(x) = \sigma(x)(1 - \sigma(x)).}$$

La dernière relation montre un autre avantage de la fonction σ : si on a calculé $\sigma(x_0)$ alors, sans effort supplémentaire, on sait calculer $\sigma'(x_0)$.
Pour mémoire, on rappelle l'allure des graphes des fonctions $x \mapsto e^x$ (à gauche) et $x \mapsto e^{-x}$ (à droite) en se souvenant que $e^{-x} = 1/e^x$.

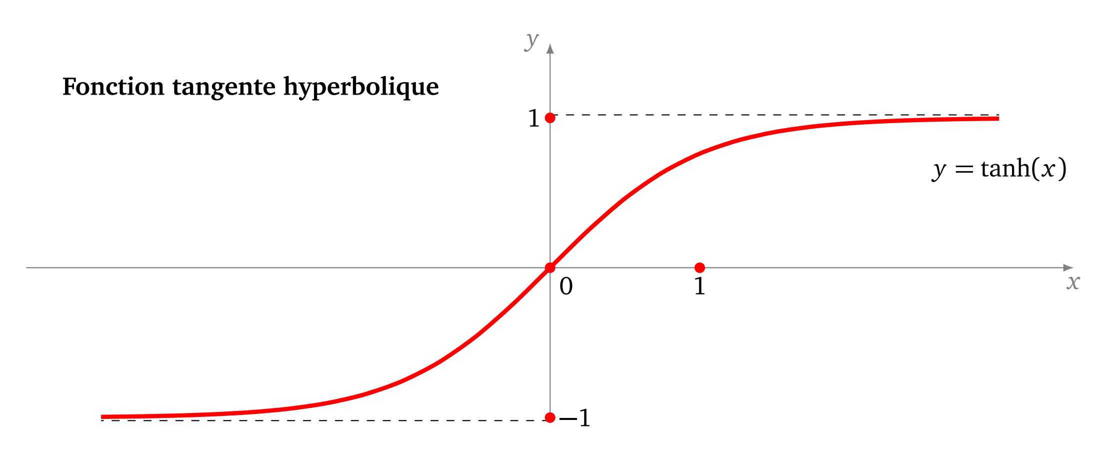


Démonstration.

1. La fonction $x \mapsto e^{-x}$ est strictement décroissante, donc la fonction $x \mapsto 1 + e^{-x}$ l'est aussi. Ainsi son inverse $x \mapsto \sigma(x) = 1/(1+e^{-x})$ est une fonction strictement croissante.
2. e^{-x} tend vers 0 en $+\infty$ donc $\sigma(x) = 1/(1+e^{-x})$ tend vers 1 lorsque x tend vers $+\infty$. De même e^{-x} tend vers $+\infty$ en $-\infty$ donc $\sigma(x)$ tend vers 0 lorsque x tend vers $-\infty$.
3. On calcule la dérivée de σ par la formule de la dérivée d'un quotient. Comme la dérivée est strictement positive cela prouve à nouveau que σ est strictement croissante.
4. Au passage, on a $\sigma'(0) = \frac{1}{4}$. La tangente en $x = 0$ a donc pour équation $y = \frac{1}{4}x + \frac{1}{2}$. Et, anticipant sur la suite, on calcule que la dérivée seconde $\sigma''(0) = 0$. La courbe possède donc un point d'inflexion en $x = 0$, ce qui justifie le caractère très linéaire du graphe autour de $x = 0$.
5. Enfin :
$$\sigma(x)\big(1-\sigma(x)\big) = \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} = \sigma'(x).$$

□

La fonction tangente hyperbolique.
La *tangente hyberbolique* est définie par :
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$



Elle a des propriétés similaires à la fonction sigmoïde mais varie de -1 à $+1$. C'est une fonction impaire : son graphe est symétrique par rapport à l'origine. On pourrait refaire une étude complète de cette fonction mais celle-ci est liée à celle de la fonction σ.

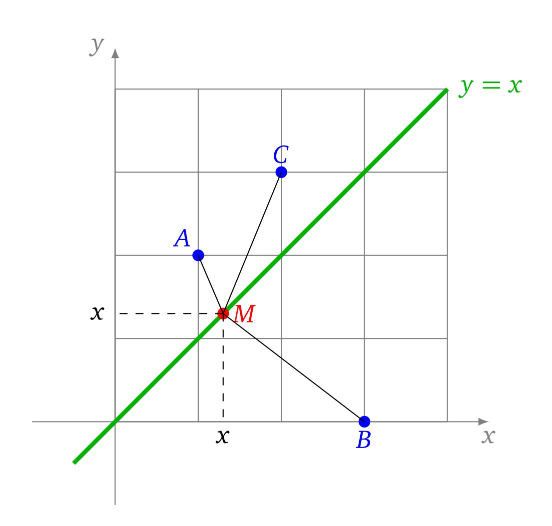
Proposition 6.

$$\boxed{\tanh(x) = 2\sigma(2x) - 1.}$$

Démonstration.
$$2\sigma(2x) - 1 = \frac{2}{1+e^{-2x}} - 1 = \frac{1-e^{-2x}}{1+e^{-2x}} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \tanh(x).$$
L'avant-dernière égalité s'obtient en multipliant numérateur et dénominateur par e^x. □

3.3. Minimum et maximum

Pour configurer un réseau de neurones, il faut fixer des paramètres réels (appelés *poids*). Les bons paramètres sont ceux qui permettent au réseau de répondre correctement à la problématique (du genre « Est-ce une photo de chat ? »). Il n'y a pas de méthode directe pour trouver immédiatement quels sont les meilleurs paramètres. Ceux-ci sont obtenus par essais/erreurs, nous y reviendrons.
Pour le moment, la question est de savoir : que sont de « bons » paramètres ?
Voyons le cas d'un seul paramètre avec le problème suivant : on se donne trois points du plan $A(1,2)$, $B(3,0)$ et $C(2,3)$. On cherche le point $M(x,x)$ qui a la contrainte d'être sur la droite d'équation $(y = x)$ et qui approche au mieux les trois points A, B et C.



Il faut préciser ce que l'on entend par « approche au mieux ». Nous souhaitons que la somme des carrés des distances entre M et chacun des points soit la plus petite possible. Nous allons définir une fonction d'erreur et chercher le point M qui minimise cette erreur. Définissons la fonction d'erreur :

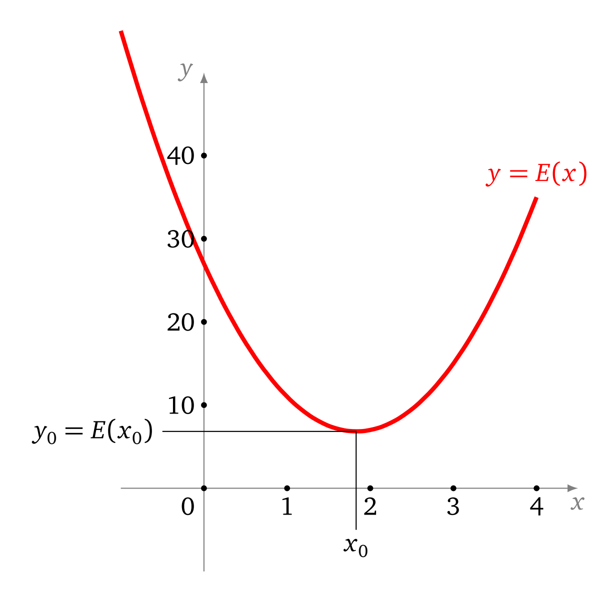
$$E(x) = AM^2 + BM^2 + CM^2.$$

La formule algébrique est la suivante :

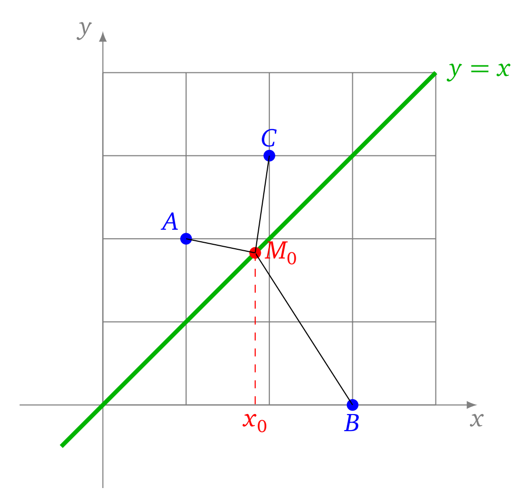
$$E(x) = \big((x-1)^2 + (x-2)^2\big) + \big((x-3)^2 + (x-0)^2\big) + \big((x-2)^2 + (x-3)^2\big).$$

Il s'agit de trouver la valeur x de sorte que $E(x)$ soit le plus petit possible. Pour cela, nous pouvons développer les termes de $E(x)$ et réécrire :

$$E(x) = ax^2 + bx + c \qquad \text{avec } a = 6,\ b = -22,\ c = 27.$$



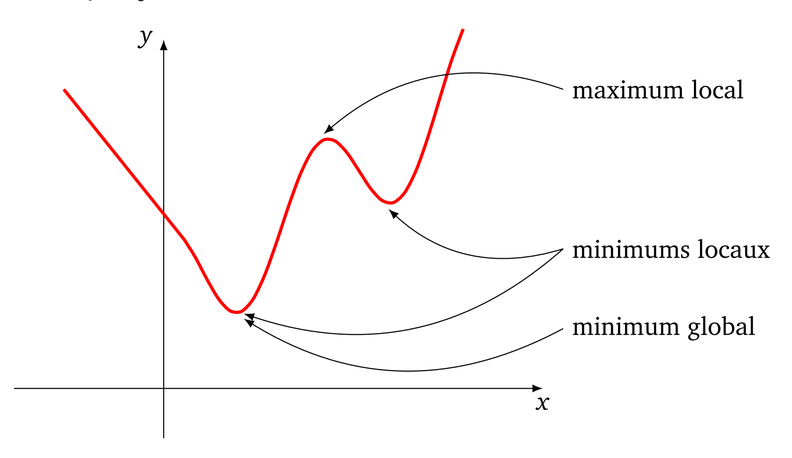
La courbe d'erreur est donc une parabole, la valeur la plus petite possible est atteinte en $x_0 = -\frac{b}{2a} = \frac{22}{12} = 1.83\ldots$ qui correspond au sommet de la parabole. Pour cette valeur x_0, on trouve $E(x_0) = \frac{41}{6} = 6.83\ldots$ et tout autre paramètre $x \in \mathbb{R}$ vérifie $E(x) \geqslant E(x_0)$.
Voici la configuration minimale avec $M_0 = (x_0, x_0)$.



De façon plus générale nous avons les définitions suivantes.

Définition.
Soit $f : \mathbb{R} \to \mathbb{R}$ une fonction.

- f atteint un ***minimum global*** en $x_0 \in \mathbb{R}$ si pour tout $x \in \mathbb{R}$, on a $f(x) \geqslant f(x_0)$.
- f atteint un ***minimum local*** en $x_0 \in \mathbb{R}$ si il existe un intervalle ouvert I, contenant x_0 et tel que pour tout $x \in I$, on a $f(x) \geqslant f(x_0)$.

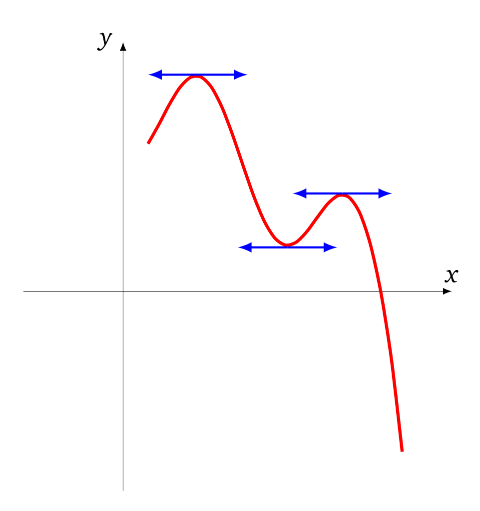


- On définirait de façon analogue un ***maximum global*** et un ***maximum local***.
- Comme son nom l'indique, pour savoir si on a un minimum local, on ne regarde que ce qui se passe autour de x_0 (l'intervalle I peut être très petit autour de x_0).
- Un minimum global est aussi un minimum local, mais l'inverse n'est pas vrai en général.

Comme on l'a vu, trouver la meilleure solution d'un problème, correspond à trouver le minimum global d'une fonction. C'est un problème difficile en général. Par contre, il est facile de trouver des valeurs qui sont les candidats à être des minimums locaux.

Proposition 7.
Si f atteint un minimum local ou un maximum local en x_0 alors $f'(x_0) = 0$.

On appellera ***point critique*** une valeur x_0 vérifiant $f'(x_0) = 0$. Voici l'interprétation géométrique : si f atteint un minimum local ou un maximum local en x_0, alors la tangente au graphe en x_0 est horizontale.



Attention ! La réciproque de cette proposition n'est pas vraie. Il se peut que $f'(x_0) = 0$ mais que f n'atteigne ni un minimum local, ni un maximum local en x_0. Nous y reviendrons.

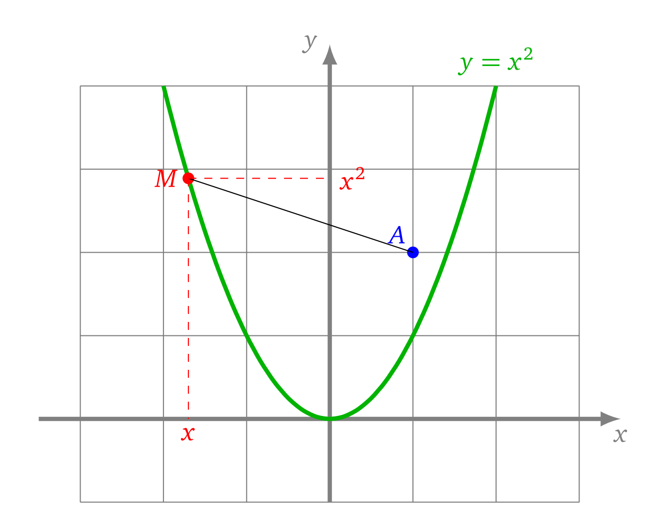
Démonstration. Prenons le cas d'un minimum local en x_0.

- Pour x proche de x_0 avec $x > x_0$, on a $f(x_0) \leqslant f(x)$, car en x_0 on atteint un minimum local. Donc le taux d'accroissement $\frac{f(x)-f(x_0)}{x-x_0}$ est positif. En passant à la limite, lorsque $x \to x_0$ avec $x > x_0$, on obtient que $f'(x_0) \geqslant 0$.
- Pour x proche de x_0 avec $x < x_0$, le taux d'accroissement $\frac{f(x)-f(x_0)}{x-x_0}$ est négatif (car cette fois $x - x_0 < 0$), à la limite, lorsque $x \to x_0$ avec $x < x_0$, on obtient que $f'(x_0) \leqslant 0$.
- Conclusion : $f'(x_0) \geqslant 0$ et $f'(x_0) \leqslant 0$, donc $f'(x_0) = 0$.

□

Reprenons l'exemple de notre fonction $E(x) = 6x^2 - 22x + 27$, et plus généralement celui d'une fonction $E(x) = ax^2 + bx + c$, alors $E'(x) = 2ax + b$. Le seul point en lequel la dérivée s'annule est $x_0 = -\frac{b}{2a}$ qui correspond bien au sommet de la parabole et est la valeur en laquelle la fonction E atteint son minimum (ou bien son maximum si on avait $a < 0$). Sur cet exemple le minimum local est aussi un minimum global.
On termine avec un autre exemple un peu plus sophistiqué.

> **Exemple.**
> Quel point de la parabole d'équation $(y = x^2)$ est le plus près du point A de coordonnées $(1, 2)$?



Les coordonnées d'un point M appartenant à cette parabole sont de la forme (x, x^2). La distance entre A et M est donc :

$$AM = \sqrt{(x-1)^2 + (x^2-2)^2}.$$

Mais trouver le plus petit AM équivaut à trouver le plus petit AM^2. On définit donc comme fonction à minimiser

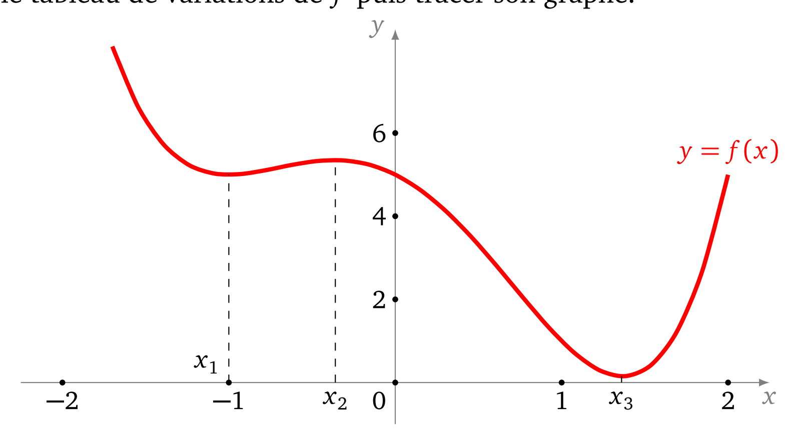
$$f(x) = AM^2 = (x-1)^2 + (x^2-2)^2 = x^4 - 3x^2 - 2x + 5.$$

Étudions cette fonction f.

- Tout d'abord $f'(x) = 4x^3 - 6x - 2$.
- Il faut chercher les valeurs en lesquelles f s'annule. Ici il se trouve que $f'(-1) = 0$, on peut donc factoriser $f'(x) = 2(x+1)(2x^2-2x-1)$. Ainsi $f'(x)$ s'annule en deux autres valeurs, les solutions de $2x^2 - 2x - 1 = 0$, c'est-à-dire en

$$x_1 = -1, \quad x_2 = \frac{1-\sqrt{3}}{2}, \quad x_3 = \frac{1+\sqrt{3}}{2}.$$

- On peut dresser le tableau de variations de f puis tracer son graphe.



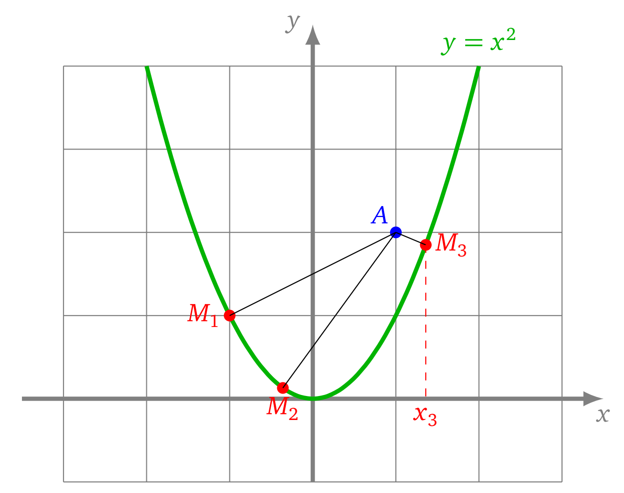
- Maximum : f admet un maximum local en x_2, mais f n'a pas de maximum global.
- Minimums : f admet un minimum local en x_1 et x_3. L'un des deux est un minimum global, mais lequel ? Il suffit de comparer $f(x_1)$ et $f(x_3)$:

$$f(x_1) = 5, \qquad f(x_3) = \frac{11-6\sqrt{3}}{4}.$$

Ainsi le minimum global est atteint en x_3 avec

$$x_3 = \frac{1+\sqrt{3}}{2} = 1.36\ldots \quad \text{et} \quad f(x_3) = 0.15\ldots$$

- Conclusion : le point de la parabole le plus proche du point A est le point M_3 d'abscisse x_3 et d'ordonnée x_3^2.



- Remarque importante. Noter que le point M_1 (correspondant à x_1) est moins éloigné de A que ses voisins proches, mais que ce n'est pas lui la solution du problème : M_1 correspond à un minimum local, mais pas un minimum global.

4. Dérivée seconde

Nous avons vu que le nombre dérivé $f'(x_0)$ permet d'approcher les valeurs de $f(x)$ pour x proche de x_0. La dérivée seconde permet de faire un peu mieux.

4.1. Définition

Soit $f : I \to \mathbb{R}$ une fonction dérivable, c'est-à-dire telle que $f'(x)$ existe pour tout $x \in I$. Nous notons f' ou $x \mapsto f'(x)$ ou $\frac{\mathrm{d}f}{\mathrm{d}x}$ la fonction dérivée.

Définition.
La ***dérivée seconde*** en x_0 (si elle existe) est la dérivée de la fonction f' en x_0, c'est-à-dire :

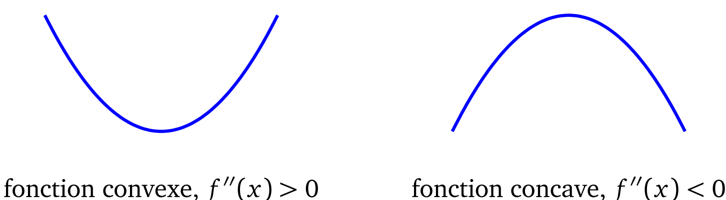
$$(f')'(x_0).$$

Nous noterons ce nombre $f''(x_0)$ ou bien $\frac{\mathrm{d}^2f}{\mathrm{d}x^2}(x_0)$.
Si la dérivée seconde existe pour tout $x \in I$, on obtient une fonction dérivée seconde notée f'' ou $x \mapsto f''(x)$ ou $\frac{\mathrm{d}^2f}{\mathrm{d}x^2}$.

Exemple.
- Soit $f(x) = x^3$. Calculons $f''(-2)$. On sait que $f'(x) = 3x^2$. La dérivée de la fonction f' est la fonction f'' définie par $f''(x) = 6x$. Donc $f''(-2) = -12$.
- Soit $f(x) = \sin(x)$ alors $f'(x) = \cos(x)$ et la dérivée seconde est $f''(x) = -\sin(x)$.
- Soit $f(x) = \exp(x^2)$, alors $f'(x) = 2x\exp(x^2)$ et $f''(x) = (2 + 4x^2)\exp(x^2)$.

4.2. Concavité et minimum local

Le signe de la dérivée seconde informe sur l'allure du graphe : si la dérivée seconde est positive sur un intervalle, alors la fonction y est convexe. La courbe est en forme de $\cup$. Le modèle est $f(x) = x^2$ (dont la dérivée seconde est $f''(x) = 2$, qui est bien positive). Si la dérivée seconde est négative sur un intervalle, alors la fonction y est concave. La courbe est en forme de $\cap$. Le modèle est $f(x) = -x^2$ (avec $f''(x) = -2$).



fonction convexe, $f''(x) > 0$ fonction concave, $f''(x) < 0$

Cela va nous aider à déterminer si un point en lequel la dérivée s'annule, c'est-à-dire un point critique, est un minimum local ou un maximum local.

Proposition 8.

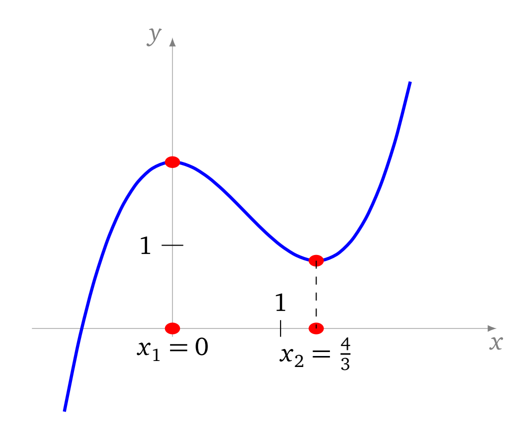
- *Si $f'(x_0) = 0$ et $f''(x_0) > 0$ alors f atteint un minimum local en x_0.*
- *Si $f'(x_0) = 0$ et $f''(x_0) < 0$ alors f atteint un maximum local en x_0.*

Si $f'(x_0) = 0$ et $f''(x_0) = 0$ alors on ne peut rien dire en général. Il faut étudier chaque fonction au cas par cas car tout peut arriver : un minimum local, un maximum local ou bien ni l'un ni l'autre.

Exemple.
Cherchons les minimums et maximums locaux de la fonction f définie par :

$$f(x) = x^3 - 2x^2 + 2.$$

- **Points critiques.** On calcule $f'(x) = 3x^2 - 4x = x(3x - 4)$ et on cherche les valeurs en lesquelles f' s'annule. Ici f' s'annule en $x_1 = 0$ et en $x_2 = 4/3$.
- **Dérivée seconde en x_1.** On calcule $f''(x) = 6x - 4$. En x_1 on a $f''(x_1) = f''(0) = -4$, donc par la proposition 8, f admet un maximum local en x_1.
- **Dérivée seconde en x_2.** En x_2 on a $f''(x_2) = f''(\frac{4}{3}) = +4$, donc par la proposition 8, f admet un minimum local en x_2.



Noter que f n'admet aucun maximum global ni aucun minimum global sur $\mathbb{R}$.

Démonstration. Prenons l'exemple d'un x_0 tel que $f'(x_0) = 0$ et $f''(x_0) > 0$. Nous supposons dans ce cours que les fonctions sont suffisamment régulières pour nos besoins. Ici nous supposons donc que f'' est continue, cela implique que $f''(x) > 0$ pour x dans un petit intervalle ouvert $]a, b[$ contenant x_0.
On peut dresser le tableau de variation de f sur $[a, b]$:

- $f''(x)$ est positive, donc f' est strictement croissante sur $[a, b]$.
- Comme f' s'annule en x_0 alors $f'(x)$ est négatif pour x dans $[a, x_0[$ et positif pour x dans $]x_0, b]$.
- Ainsi f est décroissante sur $[a, x_0]$ puis croissante sur $[x_0, b]$.
- Donc en x_0, f atteint bien un minimum local.

□

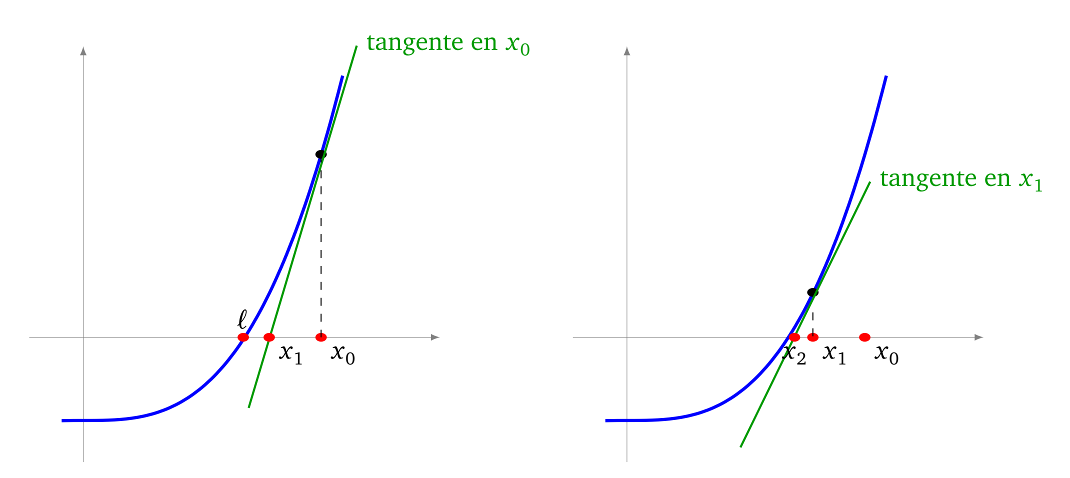
5. Méthode de Newton

La méthode de Newton est une façon efficace de trouver une solution approchée d'une équation $f(x) = 0$. Dans le paramétrage d'un réseau de neurones, il ne s'agit pas de trouver une valeur en laquelle la fonction s'annule mais une valeur en laquelle la fonction d'erreur atteint son minimum. Alors à quoi bon étudier la méthode de Newton ?
Tout d'abord, pour trouver le minimum de f, il faut identifier les points en lesquels f' s'annule. Il s'agit donc de résoudre l'équation $f'(x) = 0$ ce qui peut se faire par la méthode de Newton (appliquée à f'). Ensuite, la méthode de Newton consiste à suivre le graphe selon la direction de la tangente. C'est une idée fondamentale que l'on retrouvera plus tard lors de l'étude de la « descente de gradient » pour minimiser des fonctions de plusieurs variables.

5.1. Principe

Pour chercher une valeur approchée de ℓ de $[0, 1]$ telle que $f(\ell) = 0$, on pourrait par exemple calculer $f(0.00)$, $f(0.01)$, $f(0.02)$,...,$f(1.00)$ et choisir pour ℓ celui qui donne la valeur la plus proche de 0. C'est une méthode très lente (il faut ici évaluer 100 fois la fonction f) et peu précise (la précision sera de l'ordre 10^{-2}, soit deux chiffres exacts après la virgule).
On va voir une autre méthode plus efficace pour obtenir une valeur approchée d'une solution ℓ de $f(\ell) = 0$.
L'idée de la méthode de Newton est d'utiliser la tangente :

- on part d'une valeur x_0 quelconque,
- on trace la tangente au graphe de f au point d'abscisse x_0,
- cette tangente recoupe l'axe des abscisses en un point d'abscisse x_1 (figure de gauche),
- cette valeur x_1 est plus proche de ℓ que ne l'est x_0 (sous réserve que f satisfasse des hypothèses raisonnables),
- on recommence à partir de x_1 : on trace la tangente, elle recoupe l'axe des abscisses, on obtient une valeur x_2... (figure de droite).



Partons d'un x_0 quelconque et voyons comment calculer x_1, la valeur suivante de la suite. L'équation de la tangente en x_0 est :

$$y = (x - x_0)f'(x_0) + f(x_0).$$

On cherche le point (x_1, y_1) en lequel la tangente recoupe l'axe des abscisses. Un tel point vérifie donc l'équation de la tangente avec en plus $y_1 = 0$, ainsi :

$$0 = (x_1 - x_0)f'(x_0) + f(x_0).$$

Donc

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

5.2. Suite

On va ainsi définir par récurrence une suite (x_n). L'équation de la tangente en une valeur x_n est donnée par $y = f'(x_n)(x - x_n) + f(x_n)$.
En partant d'une valeur x_0, on obtient une formule de récurrence, pour $n \geqslant 0$:

$$\boxed{x_0 \quad \text{est donné} \qquad \text{et} \qquad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{pour } n \geqslant 0.}$$

Pour que cette méthode fonctionne, il faut tout de même partir d'une valeur x_0 pas trop éloignée de la solution ℓ cherchée.

Exemple.
On cherche une valeur approchée de $\sqrt[3]{100}$.

- Soit f définie par $f(x) = x^3 - 100$. L'unique valeur en laquelle f s'annule est bien le réel ℓ qui vérifie $\ell^3 = 100$, c'est-à-dire $\ell = \sqrt[3]{100}$.
- On calcule $f'(x) = 3x^2$.
- On part de $x_0 = 10$ par exemple.
- On calcule $f(x_0) = 900$ et $f'(x_0) = 300$. Par la formule de récurrence :
$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} = 10 - \frac{900}{300} = 7$$
- On calcule $f(x_1) = f(7) = 243$, $f'(x_1) = f'(7) = 147$ et donc
$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} = \frac{262}{49} = 5.3469\ldots$$
- Puis $x_3 = 4.7305\ldots$
- Puis $x_4 = 4.6432\ldots$
- Puis $x_5 = 4.6415894\ldots$ qui a déjà 5 chiffres après la virgule de corrects.
- Et $x_6 = 4.641588833612\ldots$ qui a 12 chiffres après la virgule de corrects.

Pour le plaisir et pour appréhender la vitesse extraordinaire de convergence, voici le terme x_{13} (après $n = 13$ itérations) qui donne 1000 chiffres exacts après la virgule pour $\sqrt[3]{100}$.

4.
64158883361277889241007635091944657655134912501124
36376506928586847778696928448261899590708975713798
41543308228265404820510270287495774377362322395030
21465094177426719650916295452146089763366938104116
28606533596551384853869619496157227826277315767548
83017169207448098556934156362916689287996611195246
16679670077293548124686871765259065725471337341531
84860446462178977769897212428546914463963789526871
82014556020545505584013855637728160923375212916394
80860747839555773984257273946686109226799406050704
02442029854177300120407410232413879663173270034106
06737496780919282092017340424063011069619562088429
61382086140688243128977537380423154514270094830453
92228536191076596869380984898548880361175285974621

5405583600665707946687248155244941081079795628903 4
2991563443219701255330594398225508295070035694296 1
3593037363732085001281953389952973264296900525800 3
4854237229598018198517132503448877442276866248182 7
7877219770507028693378636881002627302392876142571 7
3145755021578493420107450114312154988758557033486 6

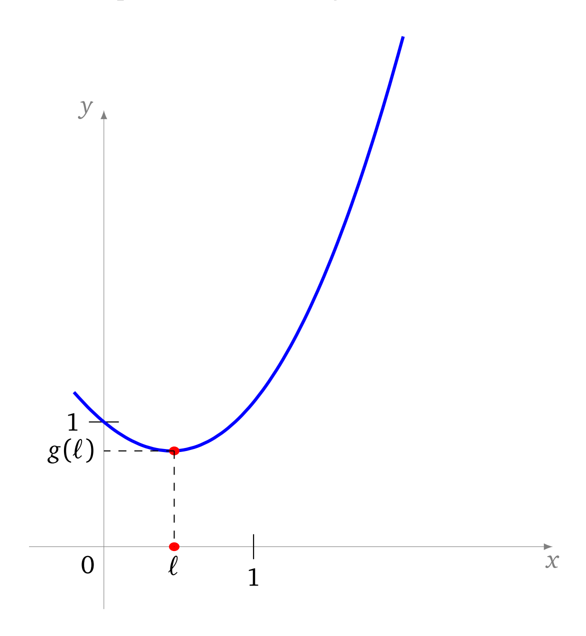
5.3. Algorithme

Algorithme de la méthode de Newton.

- — Entrée : une fonction f, une valeur de départ x_0, un nombre d'itérations n.
  — Sortie : une valeur approchée de ℓ tel que $f(\ell)=0$.
- Poser $x = x_0$.
- Répéter n fois :
$$x \leftarrow x - \frac{f(x)}{f'(x)}$$
- À la fin renvoyer x (qui approche une solution ℓ).

5.4. Exemple

Objectif. Trouver la valeur de ℓ en laquelle la fonction $g(x) = x^2 - \sin(x) + 1$ atteint son minimum.



Méthode. Il n'existe pas de méthode exacte pour trouver cette solution, nous allons donner une solution approchée de ℓ.
Calculons la dérivée de f ainsi que sa dérivée seconde :
$$g'(x) = 2x - \cos(x), \qquad g''(x) = 2 + \sin(x).$$
Une étude de fonction très simple montre que : (a) $g''(x) > 0$, la dérivée est donc strictement croissante ; (b) la dérivée ne s'annule qu'en une seule valeur ℓ qui est la solution de $g'(x) = 0$ autrement dit $\cos(x) = 2x$. On pose donc $f(x) = 2x - \cos(x)$ (c'est-à-dire $f(x) = g'(x)$) et on applique la méthode de Newton à f. Partons de $x_0 = 0$. Voici la suite obtenue et l'erreur commise à chaque étape.

$$\begin{array}{l|l} x_0 = 0 & \epsilon_0 \leqslant 1 \\ x_1 = 0.5 & \epsilon_1 \leqslant 10^{-1} \\ x_2 = 0.4506266930\ldots & \epsilon_2 \leqslant 10^{-3} \\ x_3 = 0.4501836475\ldots & \epsilon_3 \leqslant 10^{-7} \\ x_4 = 0.4501836112\ldots & \epsilon_4 \leqslant 10^{-15} \end{array}$$

Conclusion. Une valeur approchée de ℓ est donc donnée par

$$\ell \simeq 0.4501836112$$

et en ce point $f(\ell) = 0$ (donc $g'(\ell) = 0$) et g atteint sa valeur minimale :

$$g(\ell) \simeq 0.7675344248.$$

Partir d'une autre valeur, par exemple $x_0 = 10$, conduit rapidement à la même solution.

$$\begin{aligned} x_0 &= 10 \\ x_1 &= -4.3127566511\ldots \\ x_2 &= -1.4932228819\ldots \\ x_3 &= 1.5615319504\ldots \\ x_4 &= 0.5235838839\ldots \\ x_5 &= 0.4511295428\ldots \\ x_6 &= 0.4501837766\ldots \\ x_7 &= 0.4501836112\ldots \end{aligned}$$

Chapitre 2 — Python : numpy et matplotlib avec une variable

Vidéo ■ partie 2.1. Numpy
Vidéo ■ partie 2.2. Matplotlib

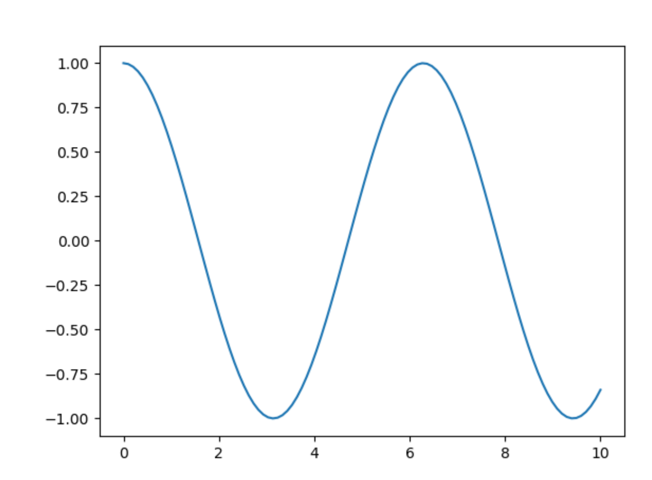
Le but de ce court chapitre est d'avoir un aperçu de deux modules Python : numpy et matplotlib. Le module numpy aide à effectuer des calculs numériques efficacement. Le module matplotlib permet de tracer des graphiques.

Ce chapitre est un chapitre plutôt technique, il suppose une connaissance de base de *Python*. Dans un premier temps, nous allons uniquement nous intéresser aux fonctions d'une variable. Il faut prendre garde à ne pas trop s'attarder sur les détails et les multiples fonctionnalités de ces deux modules. Vous pourrez revenir vers ce chapitre en fonction de vos besoins futurs.
Voici un aperçu du code le plus simple que l'on puisse écrire pour tracer une fonction. Il s'agit de tracer le graphe de la fonction $x \mapsto \cos(x)$ sur $[0, 10]$.

```
import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(0,10,num=100)
Y = np.cos(X)

plt.plot(X,Y)
plt.show()
```



Attention ! *numpy* et *matplotlib* ne sont pas toujours installés avec votre distribution *Python*. Vous devrez peut-être les installer vous-même.

1. Numpy (une variable)

On appelle le module *numpy* et on le renomme de façon raccourcie « `np` » :

```
import numpy as np
```

1.1. Définition d'un vecteur

Les principaux objets de *numpy* sont des « tableaux ». Dans ce chapitre nous étudierons seulement les tableaux à une dimension que l'on appelle des ***vecteurs***.

- Définition à partir d'une liste :

```
X = np.array([1,2,3,4])
```

- Affichage par `print(X)` :

```
[1 2 3 4]
```

Cela ressemble beaucoup à une liste, mais ce n'est pas une liste usuelle de *Python*. Le type de `X` est `numpy.ndarray`. Notez qu'il n'y a pas de virgules dans l'affichage du vecteur.

- Définition d'une suite d'éléments avec `arange()`. Cette fonction a le même comportement que la fonction classique `range()` avec en plus la possibilité d'utiliser un pas non entier. Par exemple :

```
np.arange(1,8,0.5)
```

renvoie :

```
[1.  1.5 2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5 7.  7.5]
```

- Définition d'une division d'intervalle avec `linspace()`. C'est l'une des méthodes les plus utilisées. Elle s'utilise avec la syntaxe `linspace(a,b,num=n)` pour obtenir une subdivision régulière de $[a, b]$ de n valeurs (donc en $n-1$ sous-intervalles). Exemple :

```
np.linspace(0,1,num=12)
```

vaut :

```
[0.         0.09090909 0.18181818 0.27272727 0.36363636 0.45454545
 0.54545455 0.63636364 0.72727273 0.81818182 0.90909091 1.        ]
```

1.2. Opérations élémentaires

Voici les opérations élémentaires utiles pour manipuler des vecteurs. Elles ont toutes la particularité d'agir sur les coordonnées.

Définissons un vecteur X par :

```
X = np.array([1,2,3,4])
```

- Multiplication par un scalaire. Par exemple `2*X` renvoie le vecteur `[2 4 6 8]`.
- Addition d'une constante. Par exemple `X+1` renvoie le vecteur `[2 3 4 5]`.
- Carré. Par exemple `X**2` renvoie le vecteur `[1 4 9 16]`.
- Inverse. Par exemple `1/X` renvoie le vecteur `[1.  0.5  0.33333333  0.25]`.
- Somme. Par exemple `np.sum(X)` renvoie le nombre 10.
- Minimum, maximum. Par exemple `np.min(X)` renvoie 1 et `np.max(X)` renvoie 4.

Si Y est un autre vecteur défini par `Y = np.array([10,11,12,13])` alors :

- Somme terme à terme. Par exemple `X+Y` renvoie `[11 13 15 17]`.
- Produit terme à terme. Par exemple `X*Y` renvoie `[10 22 36 52]`.

1.3. Définition d'un vecteur (suite)

Voici plusieurs façons d'initialiser un vecteur.

- Avec des zéros. Exemple : `np.zeros(5)` renvoie `[0. 0. 0. 0. 0.]`.
- Avec des uns. Exemple : `X = np.ones(5)`, alors `X` vaut `[1. 1. 1. 1. 1.]`. Ainsi `7*X` renvoie `[7. 7. 7. 7. 7.]`.
- Avec des nombres aléatoires entre 0 et 1. Exemple : `np.random.random(5)` renvoie un vecteur dont les coordonnées sont tirées aléatoirement à chaque appel, on obtient par exemple :

```
[0.21132407 0.30685886 0.94111979 0.39597993 0.63275735]
```

1.4. Utilisation comme une liste

Prenons l'exemple du vecteur X :

```
[1. 1.11111111 1.22222222 1.33333333 1.44444444 1.55555556 1.66666667
 1.77777778 1.88888889 2. ]
```

défini par la commande :

```
X = np.linspace(1,2,num=10)
```

- Récupérer une coordonnée. `X[0]` renvoie le premier élément de `X`, `X[1]` renvoie le second élément, etc.
- Longueur d'un vecteur. `len(X)` renvoie le nombre d'éléments. Une commande similaire est `np.shape(X)`.
- Parcourir tous les éléments. Deux méthodes :

```
for x in X:
    print(x)
```

```
for i in range(len(X)):
    print(i,X[i])
```

1.5. Application d'une fonction

Avec *numpy*, on peut appliquer une fonction directement sur chaque coordonnée d'un vecteur.
Prenons l'exemple de :

```
X = np.array([0,1,2,3,4,5])
```

- Racine carré. Par exemple `np.sqrt(X)` renvoie le vecteur `[0. 1. 1.41421356 1.73205081 2. 2.23606798]`. Pour chaque composante du vecteur X on a calculé sa racine carrée.
- Puissance. Par exemple `X**2` renvoie le vecteur `[0 1 4 9 16 25]`.
- Les fonctions mathématiques essentielles sont implémentées ainsi que la constante π : `np.pi`. Par exemple `np.cos(X)` (calcul des cosinus, unité d'angle le radian) ou bien `np.cos(2*np.pi/360*X)` (calcul des cosinus, unité d'angle le degré).

Pourquoi est-il est important d'utiliser `np.sqrt(X)` plutôt qu'une boucle du type `for x in X: np.sqrt(x)` ? D'une part le code est plus lisible, mais surtout les calculs sont optimisés. En effet, *numpy* profite de la présence de plusieurs processeurs (ou cœurs) dans la machine et effectue les calculs en parallèle. Les calculs sont beaucoup plus rapides.

Attention ! *numpy* se substitue au module `math` qu'il ne faut pas utiliser. (Par exemple la commande `math.cos(X)` renverrait une erreur.)

Voici les principales fonctions utiles pour ce cours. Il en existe plein d'autres !

`x ** a`	$x^a \quad (a \in \mathbb{R})$
`sqrt(x)`	$\sqrt{x}$
`exp(x)`	$\exp x$
`log(x)`	$\ln x$ logarithme népérien
`log10(x)`	$\log x$ logarithme décimal
`cos(x), sin(x), tan(x)`	$\cos x$, $\sin x$, $\tan x$ en radians
`arccos(x), arcsin(x), arctan(x)`	$\arccos x$, $\arcsin x$, $\arctan x$ en radians
`cosh(x), sinh(x), tanh(x)`	$\operatorname{ch} x$, $\operatorname{sh} x$, $\tanh x$

2. Matplotlib (une variable)

Nous allons tracer les graphes de fonctions d'une variable à l'aide des modules *matplotlib* et *numpy*.

2.1. Un exemple

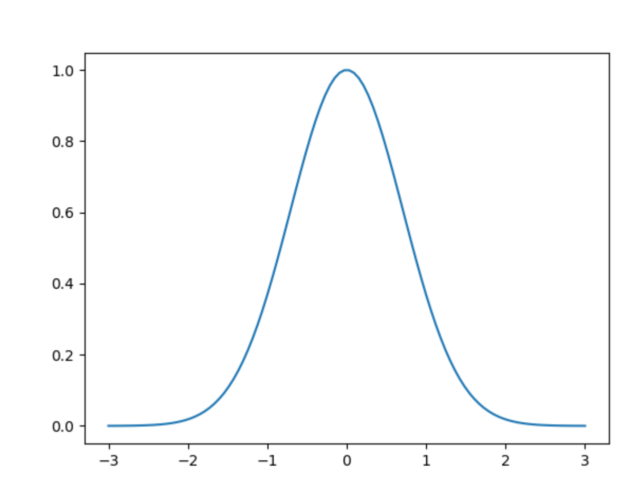
Voici comment tracer la fonction $f : [-3,3] \to \mathbb{R}$ définie par $f(x) = e^{-x^2}$, c'est la courbe de Gauss.

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.exp(-x**2)

a,b = -3,3
X = np.linspace(a,b,num=100)
Y = f(X)

plt.plot(X,Y)
plt.show()
```

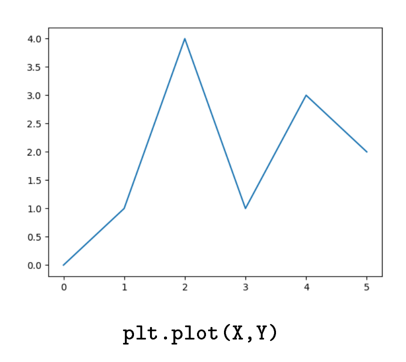


2.2. Tracé de fonctions point par point

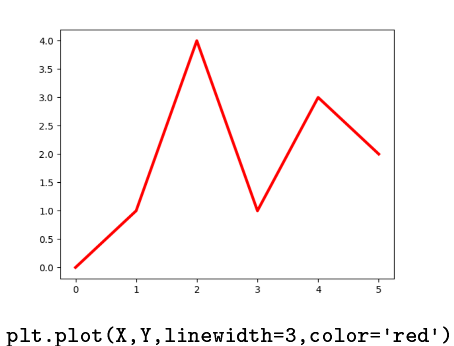
Soient deux vecteurs :

`X = np.array([0,1,2,3,4,5])` `Y = np.array([0,1,4,1,3,2])`

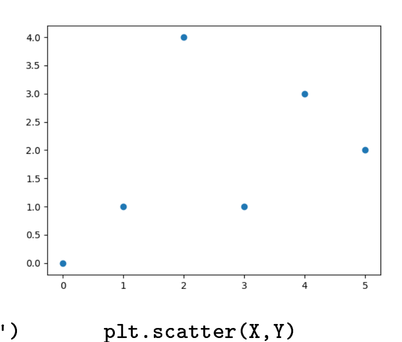
Alors la commande `plt.plot(X,Y)` affiche les points (x, y) pour x parcourant X et y parcourant Y et les relie entre eux (`plt` est le nom raccourci que nous avons donné au sous-module `pyplot` du module *matplotlib*). Lorsque les points sont suffisamment rapprochés cela donne l'impression d'une courbe lisse.



`plt.plot(X,Y)`



`plt.plot(X,Y,linewidth=3,color='red')`



`plt.scatter(X,Y)`

On peut bien sûr changer le style du tracé ou n'afficher que les points.

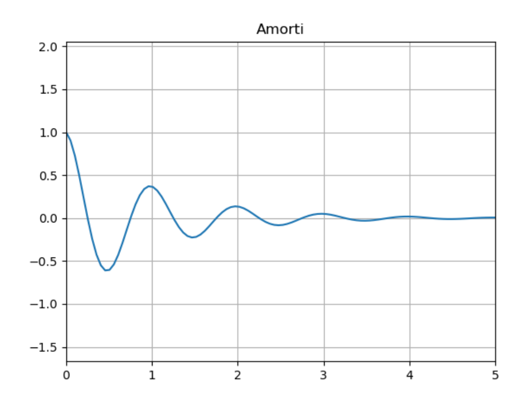
2.3. Axes

On peut exiger que le repère soit orthonormé, ajouter une grille pour plus de lisibilité. Enfin, la commande `plt.savefig()` permet de sauvegarder l'image.

```
def f(x):
    return np.exp(-x) * np.cos(2*np.pi*x)

a,b = 0,5
X = np.linspace(a,b,num=100)
Y = f(X)

plt.title('Amorti') # titre
plt.axis('equal')   # repère orthonormé
plt.grid()          # grille
plt.xlim(a,b)       # bornes de l'axe des x
plt.plot(X,Y)
plt.savefig('amorti.png')
plt.show()
```

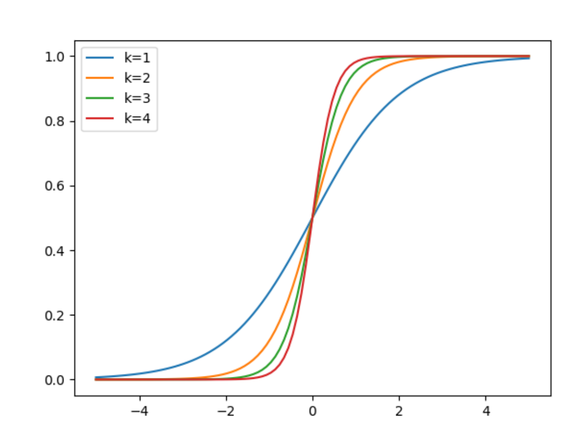


2.4. Plusieurs tracés

On peut effectuer plusieurs tracés sur le même dessin. Ici des fonctions sigmoïdes

$$f_k(x) = \frac{1}{1+e^{-kx}}$$

avec un paramètre k qui varie.




```
def f(x,k):
    return 1/(1+np.exp(-k*x))

a,b = -5,5
X = np.linspace(a,b,num=100)

for k in range(1,5):
    Y = f(X,k)
    plt.plot(X,Y, label="k={}".format(k))

plt.legend()
plt.show()
```

Chapitre 3

Fonctions de plusieurs variables

Vidéo ■ partie 3.1. Définition et graphe
Vidéo ■ partie 3.2. Minimums et maximums
Vidéo ■ partie 3.3. Recherche d'un minimum

Dans ce chapitre, nous allons nous concentrer sur les fonctions de deux variables et la visualisation de leur graphe et de leurs lignes de niveau. La compréhension géométrique des fonctions de deux variables est fondamentale pour assimiler les techniques qui seront rencontrées plus tard avec un plus grand nombre de variables.

De nombreux phénomènes dépendent de plusieurs paramètres, par exemple le volume d'un gaz dépend de la température et de la pression ; l'altitude z d'un terrain dépend des coordonnées (x, y) du lieu.
Dans les réseaux de neurones, les fonctions de plusieurs variables interviennent de deux manières :

- lors de l'utilisation d'un réseau. C'est la partie la plus facile et la plus fréquente. On utilise un réseau déjà bien paramétré pour répondre à une question (Est-ce une photo de chat ? Tourner à droite ou à gauche ? Quel pion déplacer au prochain coup ?). La réponse est un calcul direct obtenu en évaluant une fonction de plusieurs variables.
- lors de la paramétrisation du réseau. C'est la partie difficile et le but de ce cours. Quels paramètres choisir pour définir ce réseau afin qu'il réponde au problème ? Ces paramètres seront choisis comme minimum d'une fonction de plusieurs variables. Une des difficultés est qu'il peut y avoir des milliers de paramètres à gérer.

1. Définition et exemples

1.1. Définition

Nous allons étudier les fonctions de deux variables, mais aussi de trois variables et plus généralement de n variables. Ces fonctions sont donc de la forme

$$f : \mathbb{R}^n \to \mathbb{R}$$

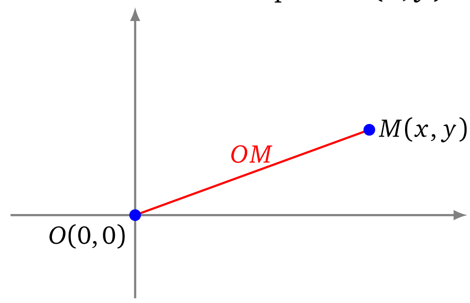
où n est un entier naturel supérieur ou égal à 1.
Un élément de l'ensemble de départ est un vecteur de type $x = (x_1, \ldots, x_n)$. À chacun de ces vecteurs, f associe un nombre réel $f(x_1, \ldots, x_n)$. On pourrait aussi limiter l'ensemble de départ à une partie E de $\mathbb{R}^n$.

1.2. Deux variables

Lorsque $n = 2$, on préfère noter les variables (x, y) plutôt que (x_1, x_2). Voici quelques exemples.

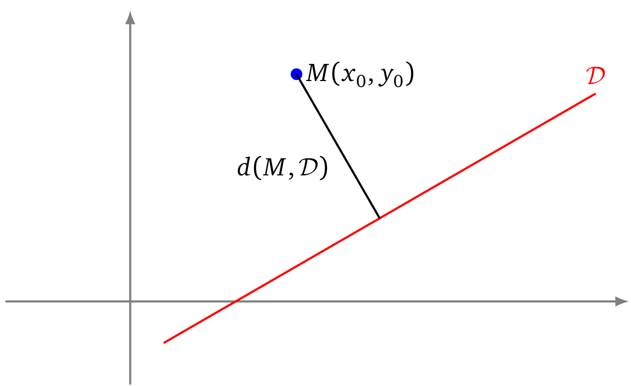
Exemple.
- $f(x, y) = 2x + 3y^2 + 1$.

- $f(x,y) = \cos(xy)$.
- $f(x,y) = \sqrt{x^2+y^2}$. f renvoie la distance entre un point $M(x,y)$ et l'origine $O(0,0)$.



- L'équation physique $PV = nRT$ implique $T = \frac{1}{nR}PV$: la température d'un gaz s'exprime en fonction de son volume et de la pression (n et R sont des constantes).
- La distance entre une droite fixée $\mathcal{D}$ d'équation $ax+by+c=0$ et un point $M(x_0,y_0)$ est donnée par une fonction de deux variables :

$$d(x_0,y_0) = d(M,\mathcal{D}) = \frac{|ax_0+by_0+c|}{\sqrt{a^2+b^2}}.$$

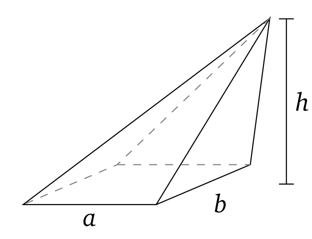


1.3. Trois variables

Exemple.

1. $f(x,y,z) = ax+by+cz+d$. Cette fonction f est une fonction affine (a,b,c,d sont des constantes).
2. $f(x,y,z) = x^2+y^2+z^2$ qui donne la distance au carré entre un point $M(x,y,z)$ et l'origine $O(0,0,0)$.
3. Le volume d'un cône à base rectangulaire dépend des longueurs des côtés a et b de la base et de la hauteur h :

$$V = f(a,b,h) = \frac{1}{3}abh.$$



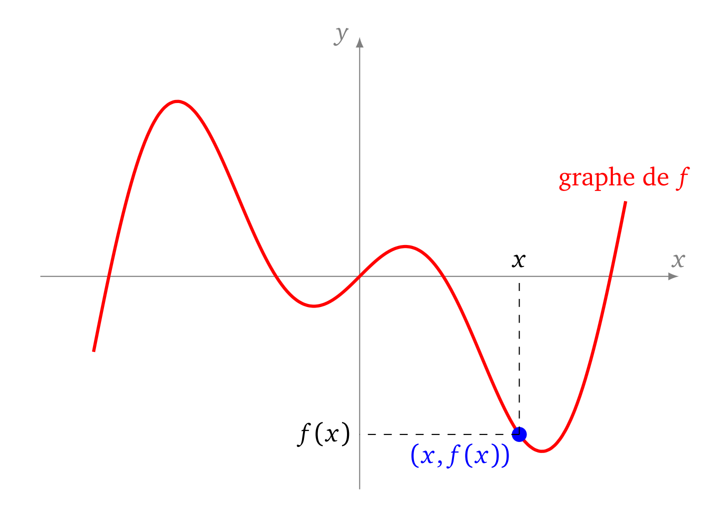
1.4. n variables

Exemple.

1. $f(x_1,\ldots,x_n) = a_1x_1 + \cdots + a_nx_n + a_0$ une fonction affine (les a_i sont des constantes).
2. $f(x_1,\ldots,x_n) = \sqrt{(x_1-a_1)^2+\cdots+(x_n-a_n)^2}$ exprime la distance entre les points $M(x_1,\ldots,x_n)$ et $A(a_1,\ldots,a_n)$ dans $\mathbb{R}^n$.

2. Graphe

Le cas le plus simple, et déjà connu, est celui des fonctions d'une seule variable $f : \mathbb{R} \to \mathbb{R}$. C'est l'ensemble de tous les points du plan de la forme $(x, f(x))$. Voici le graphe de la fonction $x \mapsto x\cos x$.

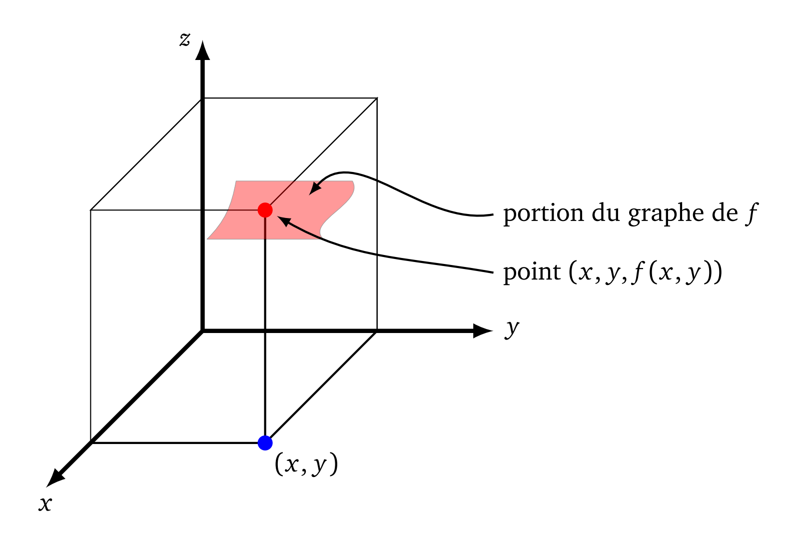


2.1. Définition

Définition.
Le ***graphe*** $\mathcal{G}_f$ d'une fonction de deux variables $f : \mathbb{R}^2 \to \mathbb{R}$ est l'ensemble des points de $\mathbb{R}^3$ ayant pour coordonnées $(x, y, f(x, y))$, pour (x, y) parcourant $\mathbb{R}^2$. Le graphe est donc :

$$\mathcal{G}_f = \left\{(x,y,z) \in \mathbb{R}^3 \mid (x,y) \in \mathbb{R}^2 \text{ et } z = f(x,y)\right\}.$$

Dans le cas de deux variables, le graphe d'une fonction est une surface tracée dans l'espace.



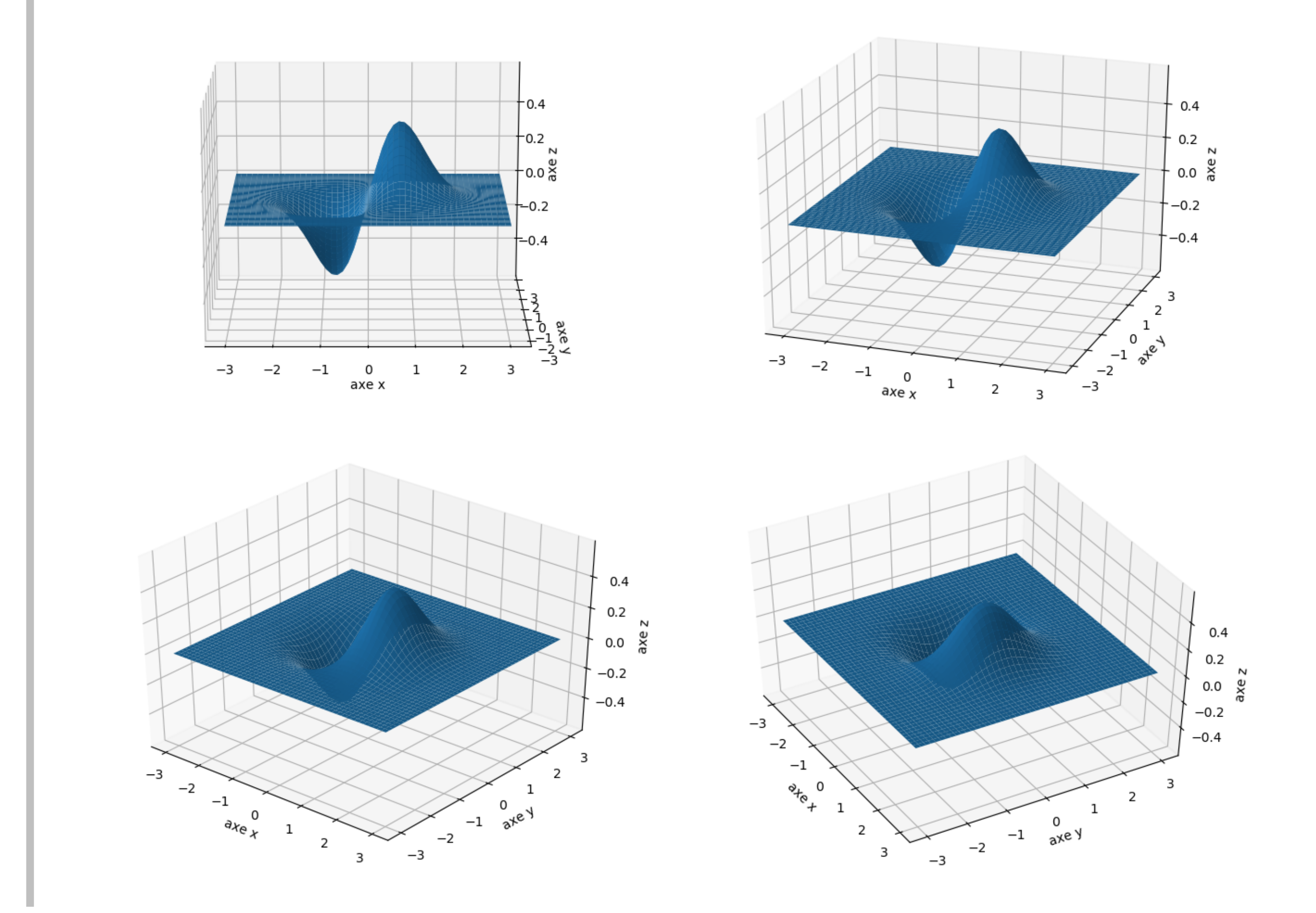
Exemple.
On souhaite tracer le graphe de la fonction définie par :

$$f(x,y) = xe^{-x^2-y^2}.$$

On commence par tracer quelques points à la main :

- si $(x,y) = (0,0)$ alors $f(x,y) = f(0,0) = 0$ donc le point de coordonnées $(0,0,0)$ appartient au graphe.
- Comme $f(1,0) = 1/e$ alors le point de coordonnées $(1,0,1/e)$ appartient au graphe.
- Pour n'importe quel y, on a $f(0,y) = 0$ donc la droite de l'espace d'équation $(x = 0$ et $z = 0)$ est incluse dans le graphe.
- Notons $r = \sqrt{x^2+y^2}$ la distance entre le point de coordonnées (x,y) et l'origine $(0,0)$ alors on a la formule $f(x,y) = xe^{-r^2}$. Pour un point éloigné de l'origine, r est grand, donc e^{-r^2} est très petit, et $f(x,y)$ est très proche de 0.

Voici différentes vues de ce graphe.



2.2. Tranches

Afin de tracer le graphe d'une fonction de deux variables, on peut découper la surface en « tranches ». On fixe par exemple une valeur y_0 et on trace dans le plan (xOz) le graphe de la fonction d'une variable

$$f_{|y_0} : x \mapsto f(x,y_0).$$

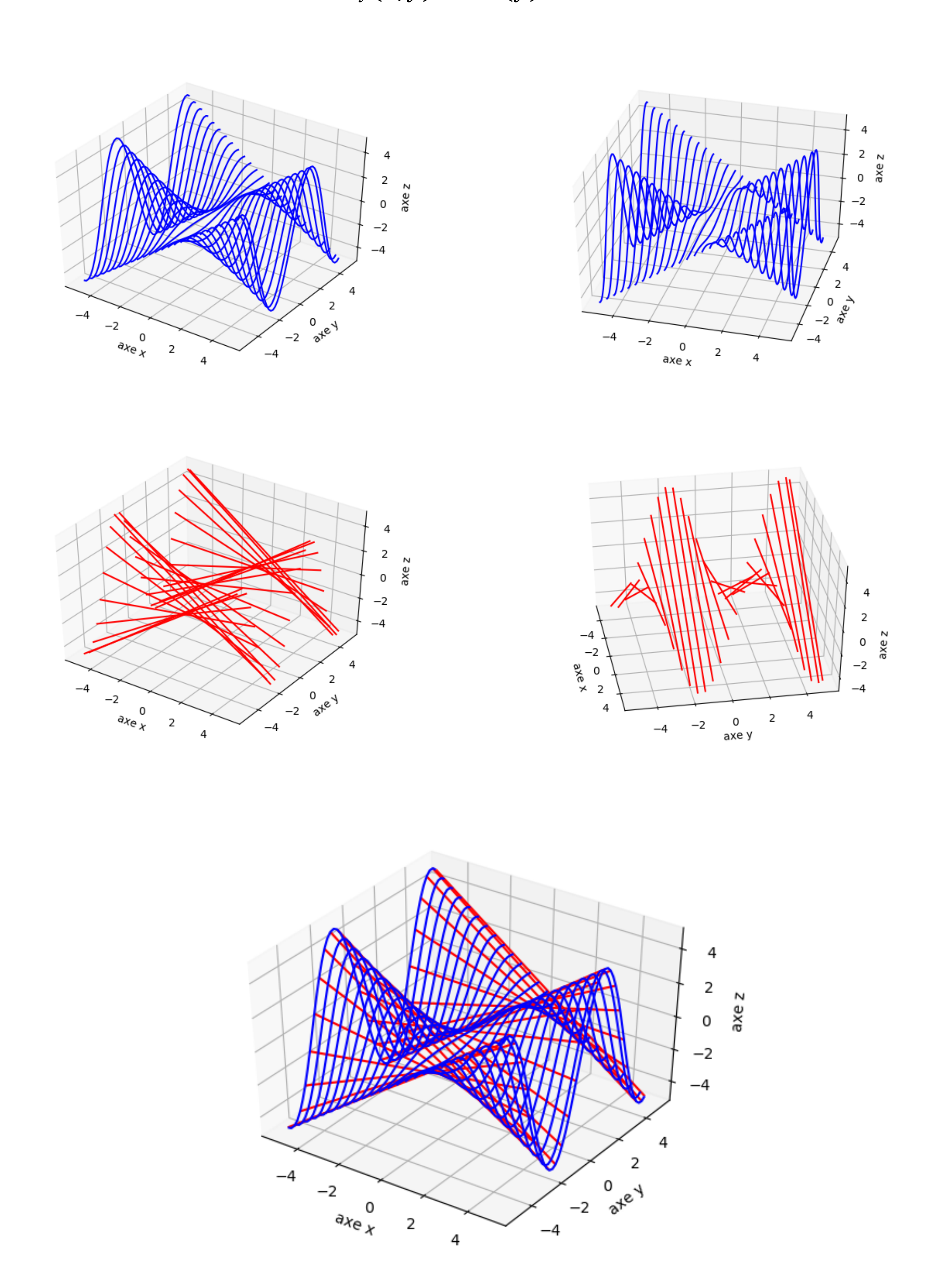
Géométriquement, cela revient à tracer l'intersection du graphe de f et du plan d'équation $(y = y_0)$. On recommence pour plusieurs valeurs de y_0, ce qui nous donne des tranches du graphe de f et nous donne une bonne idée du graphe complet de f.

On peut faire le même travail en fixant des valeurs x_0 avec les fonctions :

$$f_{|x_0} : y \mapsto f(x_0, y).$$

Exemple.
On souhaite tracer le graphe de la fonction définie par :

$$f(x, y) = x\sin(y).$$



En bleu les tranches pour lesquelles x est constant (deux points de vue), en rouge les tranches pour lesquelles y est constant (deux points de vue). Lorsque l'on rassemble les tranches (à x constant et à y constant), on reconstitue la surface (dernière figure).

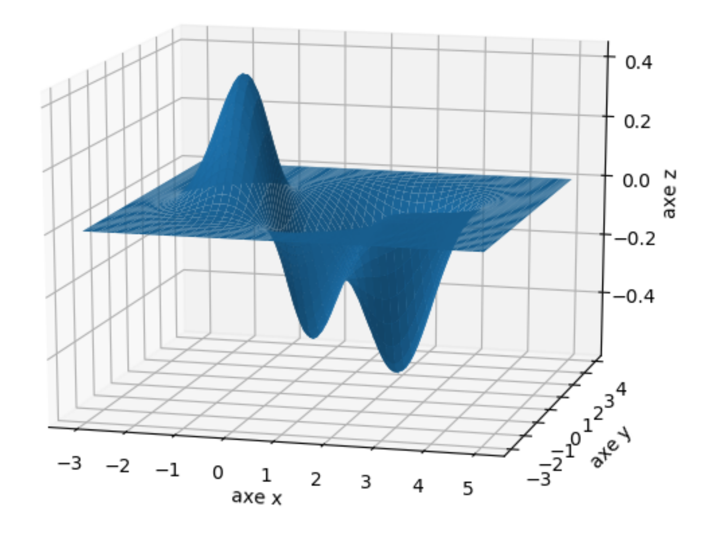
2.3. Minimum, maximum

Pour des fonctions de deux variables (ou plus) il existe une notion de minimum et de maximum.

Définition.
Soit $f : \mathbb{R}^2 \to \mathbb{R}$ une fonction.

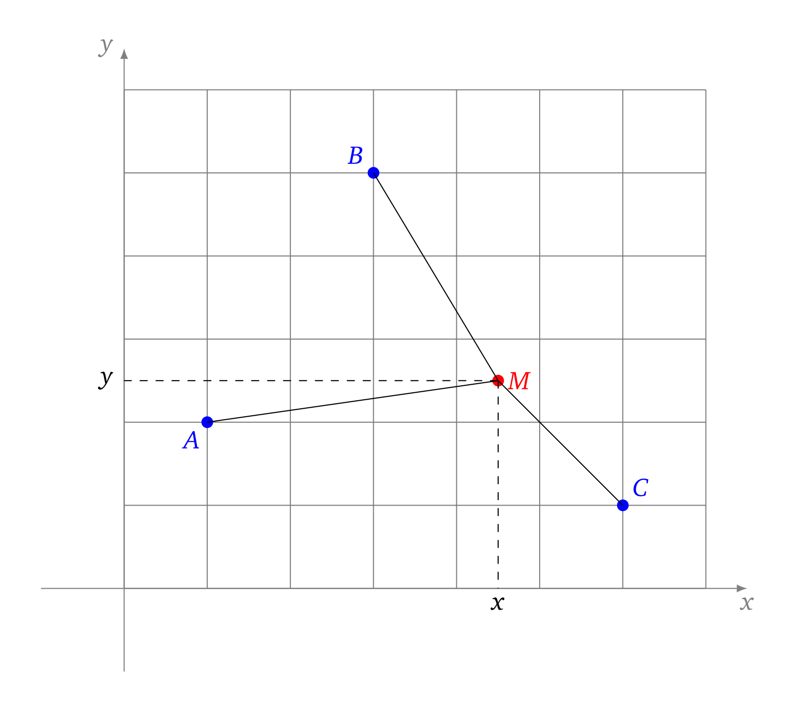
- f atteint un ***minimum global*** en $(x_0, y_0) \in \mathbb{R}^2$ si pour tout $(x, y) \in \mathbb{R}^2$, on a $f(x, y) \geqslant f(x_0, y_0)$.
- f atteint un ***minimum local*** en $(x_0, y_0) \in \mathbb{R}^2$ si il existe un intervalle ouvert I contenant x_0 et un intervalle ouvert J contenant y_0 tels que pour tout $(x, y) \in I \times J$, on a $f(x, y) \geqslant f(x_0, y_0)$.

Exemple.
Voici l'exemple d'une fonction qui admet deux minimums locaux. L'un est aussi un minimum global. Elle admet un maximum local qui est aussi global.



Trouver les bons paramètres d'un réseau de neurones nous amènera à trouver le minimum d'une fonction de plusieurs (et même de centaines de) variables. Voyons un exemple en deux variables.

Exemple.
Étant donnés trois points $A(1, 2)$, $B(3, 5)$ et $C(6, 1)$, il s'agit de trouver un point $M(x, y)$ qui « approche au mieux » ces trois points. Il faut expliciter une fonction à minimiser pour définir correctement le problème. Nous décidons de prendre la somme des carrés des distances.



Il s'agit donc de minimiser la fonction f suivante, qui correspond à une fonction distance (aussi appelée fonction erreur ou bien fonction coût) :

$$f(x,y) = MA^2 + MB^2 + MC^2 = (x-1)^2 + (y-2)^2 + (x-3)^2 + (y-5)^2 + (x-6)^2 + (y-1)^2.$$

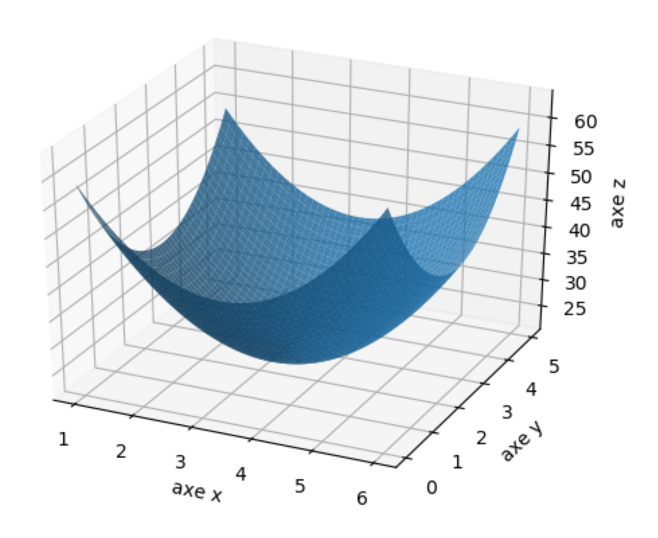
En développant on trouve :

$$f(x,y) = 3x^2 + 3y^2 - 20x - 16y + 76.$$

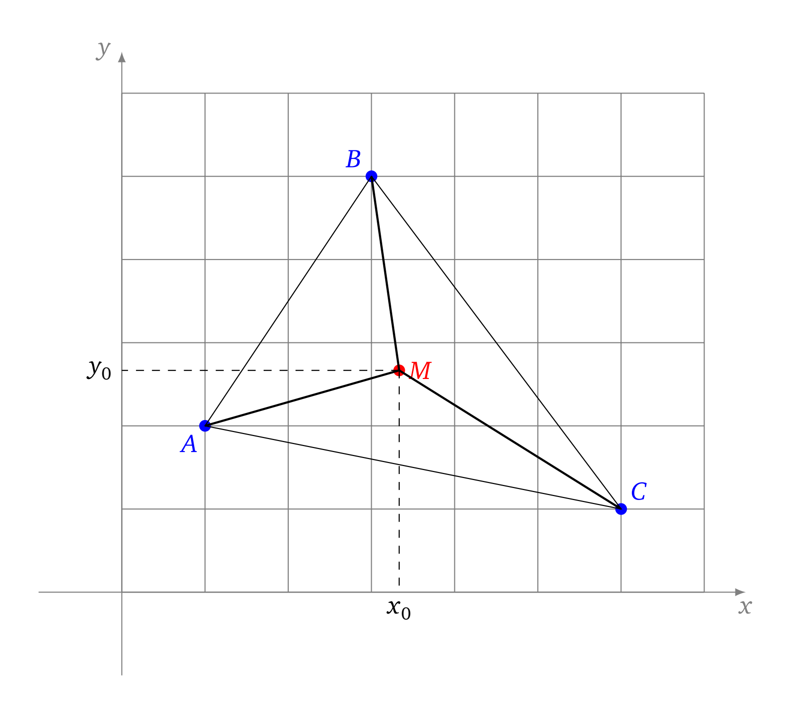
Le graphe de f nous suggère qu'il existe un unique minimum qui est le minimum global de f. Par recherche graphique ou par les méthodes décrites dans la section suivante, on trouverait une solution approchée. En fait la solution géométrique exacte est l'isobarycentre des points (autrement dit le centre de gravité du triangle ABC), ainsi :

$$(x_0, y_0) = \left(\frac{10}{3}, \frac{8}{3}\right) \simeq (3.33, 2.66)$$

pour lequel f atteint son minimum $z_0 = f(x_0, y_0) = \frac{64}{3} \simeq 21.33$.



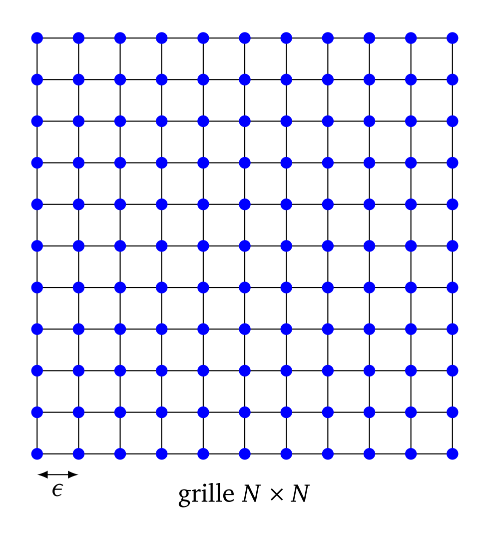
Le point qui convient le mieux à notre problème et en lequel notre fonction distance est minimale est donc le point de coordonnées $(\frac{10}{3}, \frac{8}{3})$. Attention, un autre choix de la fonction distance f pourrait conduire à une autre solution (voir l'exemple de la prochaine section).



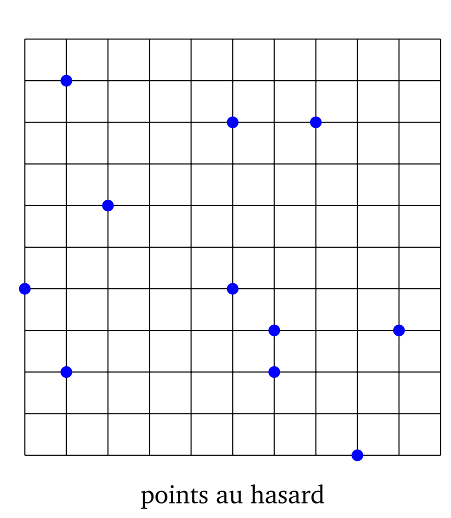
2.4. Recherche élémentaire d'un minimum

Voici trois techniques pour trouver les valeurs approchées des coordonnées du point en lequel une fonction de plusieurs variables atteint son minimum. Ces techniques sont valables quelque soit le nombre de variables même si ici elles ne sont décrites que dans le cas de deux variables.

Recherche sur une grille. On calcule $f(x, y)$ pour (x, y) parcourant une grille. On retient le point (x_0, y_0) en lequel $z_0 = f(x_0, y_0)$ est le plus petit. Si on le souhaite, on peut affiner la grille autour de ce point, en diminuant le pas ϵ pour améliorer l'approximation. C'est une technique qui demande N^2 calculs pour une grille de largeur N (et même N^n pour une fonction de n variables) ce qui peut être énorme.



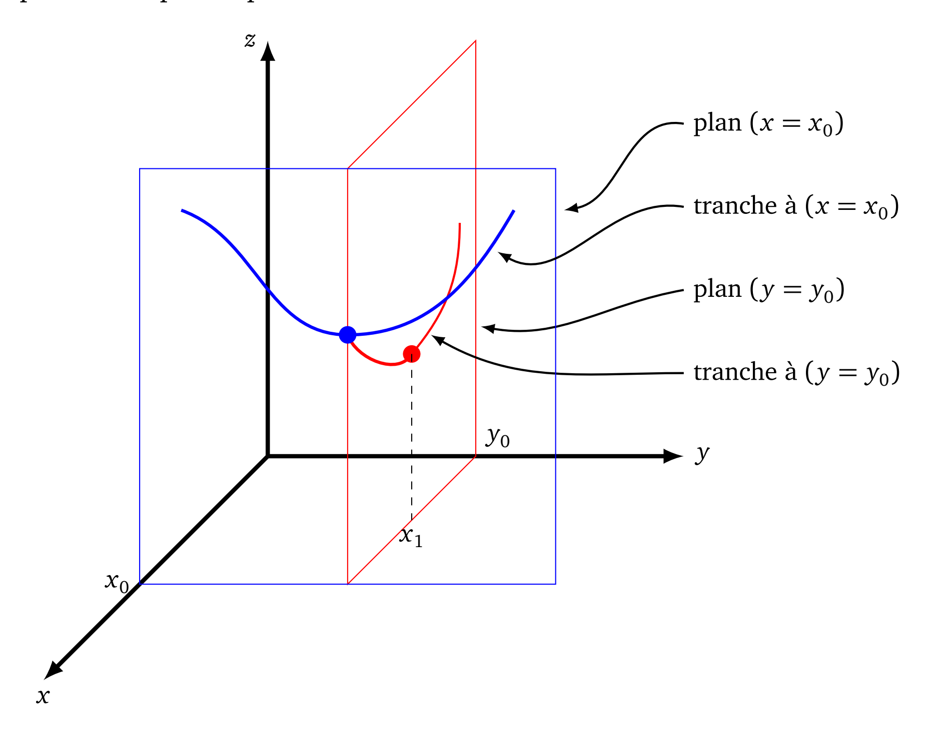
grille $N \times N$


points au hasard

Recherche au hasard. Cela peut sembler incongru mais choisir quelques coordonnées (x, y) au hasard, calculer chaque valeur $z = f(x, y)$ et comme auparavant retenir le point (x_0, y_0) correspondant au z_0 minimal n'est pas ridicule ! Un ordinateur peut tester plusieurs millions de points en quelques secondes.

Bien sûr il y a de fortes chances de ne trouver qu'une solution approchée. C'est aussi une technique que l'on retrouvera plus tard : partir d'un point au hasard pour ensuite construire une suite de points convergeant vers un minimum. Et si cela n'est pas concluant, il faudra repartir d'un autre point tiré au hasard.

Recherche par tranche. L'idée est de se ramener à des fonctions d'une seule variable. En effet, pour les fonctions d'une variable, on sait qu'il faut chercher les minimums là où la dérivée s'annule.
On part d'une valeur x_0 (au hasard !). On cherche le minimum sur la tranche $x = x_0$, c'est-à-dire que l'on cherche le minimum de la fonction d'une variable $y \mapsto f(x_0, y)$. On trouve une valeur y_0 qui réalise un minimum. On change alors de direction en étudiant maintenant la tranche $y = y_0$ (pour le y_0 que l'on vient d'obtenir), on obtient l'abscisse x_1 du minimum de la fonction d'une variable $x \mapsto f(x, y_0)$. On recommence depuis le début à partir de ce x_1. On obtient ainsi une suite de points (x_i, y_i) avec des valeurs $z_i = f(x_i, y_i)$ de plus en plus petites. On peut espérer tendre vers un minimum.



Sur la figure ci-dessus, on part d'une tranche (en bleu) choisie au hasard, donnée par $x = x_0$. Cette tranche définit le graphe d'une fonction d'une variable. On se déplace sur cette courbe jusqu'à atteindre le minimum de cette tranche, en une valeur y_0. On considère la tranche perpendiculaire donnée par $y = y_0$. On se déplace sur la courbe rouge jusqu'à atteindre le minimum de cette tranche, en une valeur x_1. On pourrait continuer avec une nouvelle tranche bleue $x = x_1$, etc.

Les descriptions données ici sont assez informelles, d'une part parce qu'il est difficile d'énoncer des théorèmes qui garantissent d'atteindre un minimum local et d'autre part parce qu'aucune technique ne garantit d'atteindre un minimum global. Lorsque l'on étudiera le gradient, nous obtiendrons une méthode plus efficace.

Exemple.
On reprend le problème précédent, à savoir trouver un point M qui approche au mieux les trois points A, B et C, mais cette fois on choisit la « vraie » distance comme fonction d'erreur :

$$g(x, y) = MA + MB + MC = \sqrt{(x-1)^2 + (y-2)^2} + \sqrt{(x-3)^2 + (y-5)^2} + \sqrt{(x-6)^2 + (y-1)^2}.$$

On applique ces trois techniques pour chercher le minimum de g en se limitant au carré $[0, 6] \times [0, 6]$.

1. Avec une grille $N \times N$. Par exemple pour $N = 100$, on évalue g en $10\,000$ points. On trouve $(x_{\min}, y_{\min}) \simeq (2.91, 2.97)$ pour une valeur $z_{\min} \simeq 7.84$.

2. Avec un tirage aléatoire de 1000 points, on trouve par exemple : $(x_{\min}, y_{\min}) \simeq (2.88, 2.99)$ et $z_{\min} \simeq 7.84$. Le résultat est similaire à la méthode précédente, bien qu'on ait effectué 10 fois moins de calculs.
3. Par les tranches. On part de la tranche $(x = 0)$. On pose $x_0 = 0$, la fonction à étudier est donc
$$g_{|x_0}(y) = \sqrt{1 + (y-2)^2} + \sqrt{9 + (y-5)^2} + \sqrt{36 + (y-1)^2}.$$
C'est une fonction de la seule variable y pour laquelle on possède des techniques efficaces de recherche de minimum. On trouve que le minimum de $g_{|x_0}$ est atteint en $y_0 \simeq 2.45$. On recommence avec cette fois la tranche $(y = y_0)$ et on cherche le minimum de la fonction $g_{|y_0}(x) = g(x, y_0)$. On trouve que cette fonction atteint son minimum en $x_1 \simeq 2.84$. On recommence ce processus jusqu'à atteindre la précision souhaitée. Ainsi en 5 étapes on obtient une valeur approchée assez précise du minimum $(x_{\min}, y_{\min}) \simeq (2.90579, 2.98464)$ et $z_{\min} \simeq 7.83867$.

La première conclusion à tirer de ce qui précède est que pour résoudre un problème il faut définir correctement une fonction d'erreur, c'est-à-dire celle que l'on cherche à minimiser. La solution trouvée dépend de cette fonction d'erreur choisie. Enfin, la méthode des tranches est une méthode efficace pour trouver un minimum d'une fonction, mais nous en découvrirons une encore meilleure en utilisant le gradient.

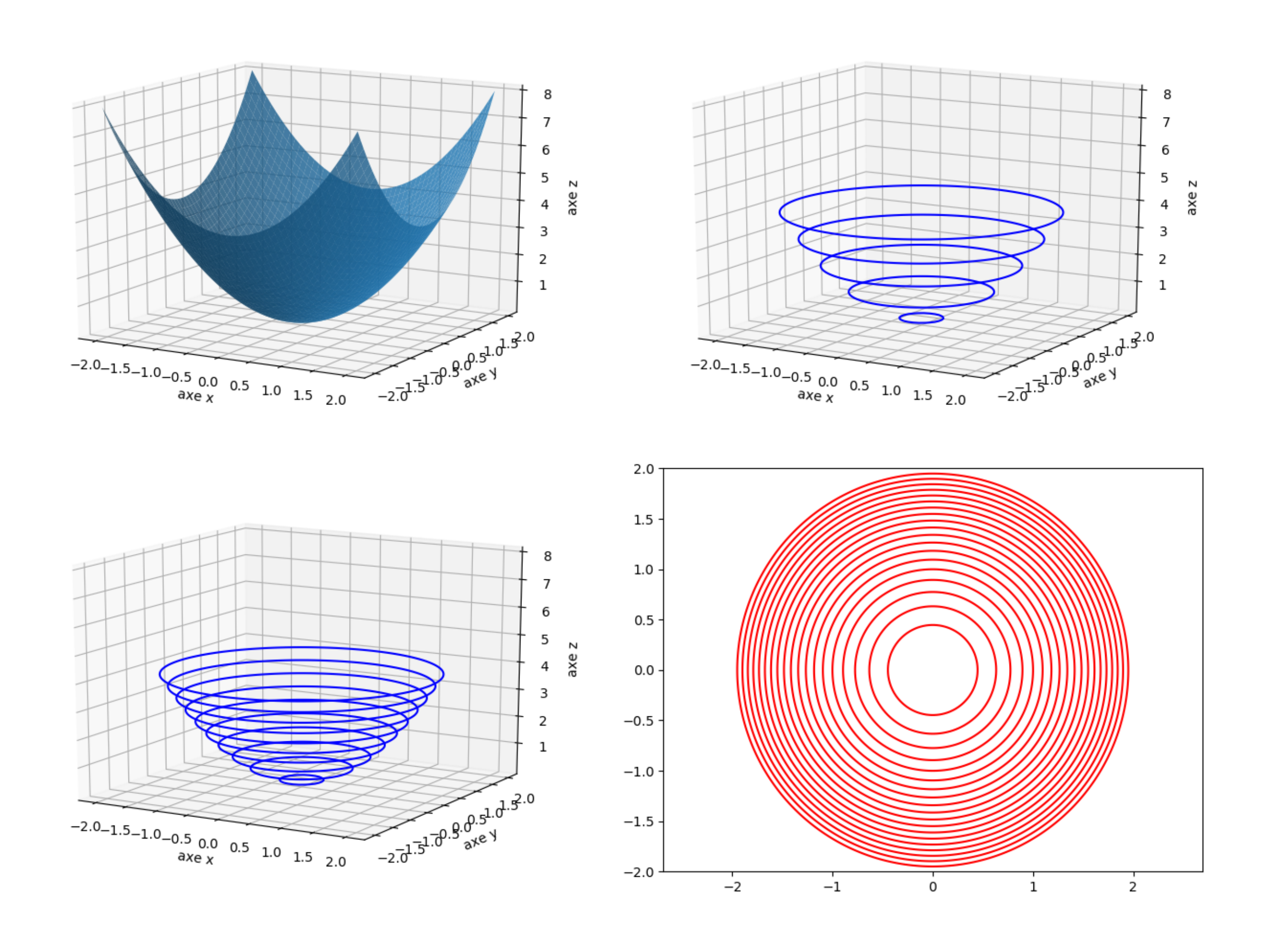
3. Lignes de niveau

3.1. Définition

Définition.
Soit $f : \mathbb{R}^2 \to \mathbb{R}$ une fonction de deux variables. La ***ligne de niveau*** $z = c \in \mathbb{R}$ est l'ensemble de tous les points (x, y) vérifiant $f(x, y) = c$:
$$L_c = \left\{(x, y) \in \mathbb{R}^2 \mid f(x, y) = c\right\}.$$
La ligne de niveau c est une courbe du plan $\mathbb{R}^2$.
On peut aussi définir une ***courbe de niveau***, c'est l'ensemble des points de l'espace obtenus comme intersection du graphe $\mathcal{G}_f$ et du plan $z = c$ qui est horizontal et « d'altitude » c. Ce sont donc tous les points $(x, y, f(x, y))$ avec $f(x, y) = c$. On obtient la courbe de niveau en translatant la ligne de niveau d'une altitude c.

Exemple.
Soit $f : \mathbb{R}^2 \to \mathbb{R}$ définie par $f(x, y) = x^2 + y^2$.
- Si $c < 0$, la ligne de niveau L_c est vide (aucun point n'est d'altitude négative).
- Si $c = 0$, la ligne de niveau L_0 se réduit à $\{(0, 0)\}$.
- Si $c > 0$, la ligne de niveau L_c est le cercle du plan de centre $(0, 0)$ et de rayon $\sqrt{c}$. On « remonte » L_c à l'altitude $z = c$: la courbe de niveau est alors le cercle horizontal de l'espace de centre $(0, 0, c)$ et de rayon $\sqrt{c}$.

Le graphe est alors une superposition de cercles horizontaux de l'espace de centre $(0, 0, c)$ et de rayon $\sqrt{c}$, $c > 0$.
Ci-dessous : (a) la surface, (b) 5 courbes de niveau, (c) 10 courbes de niveau, (d) les lignes de niveau dans le plan.



3.2. Exemples

Exemple.
Sur une carte topographique, les lignes de niveau représentent les courbes ayant la même altitude.



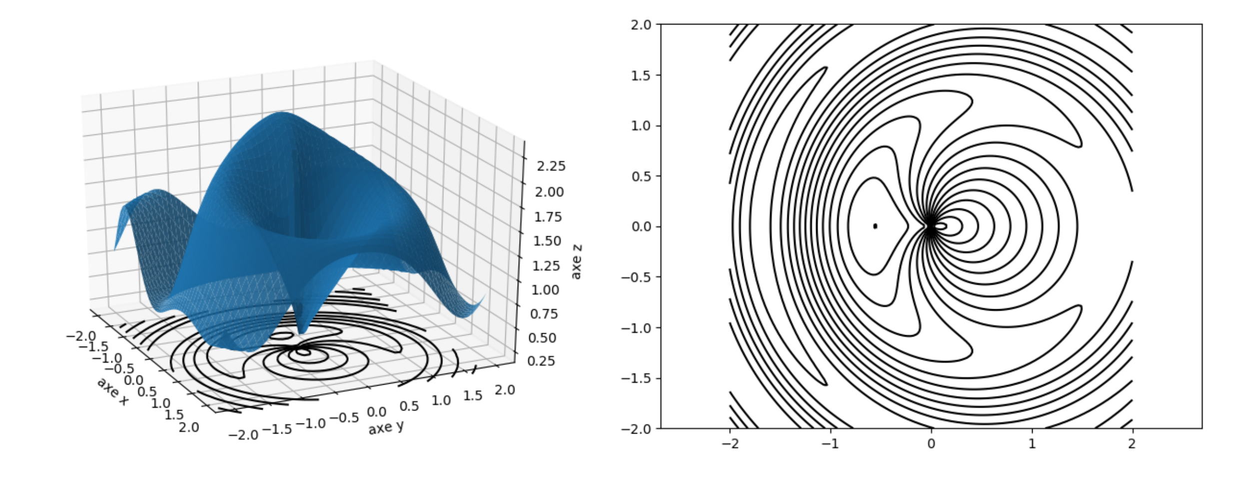
- Ici, une carte *Open Street Map* avec au centre le mont Gerbier de Jonc (source de la Loire, 1551 m).
- Les lignes de niveau correspondent à des altitudes équidistantes de 10 m (par exemple, pour $c = 1400$,

$c = 1410$, $c = 1420\ldots$).

- Lorsque les lignes de niveau sont très espacées, le terrain est plutôt plat ; lorsque les lignes sont rapprochées le terrain est pentu.
- Par définition, si on se promène en suivant une ligne de niveau, on reste toujours à la même altitude !

Exemple.
Voici le graphe et les lignes de niveau de la fonction $f : \mathbb{R}^2 \to \mathbb{R}$ définie par

$$f(x,y) = \frac{\sin(r^2 - x)}{r} + 1 \qquad \text{où } r = \sqrt{x^2 + y^2}.$$



3.3. Surfaces quadratiques

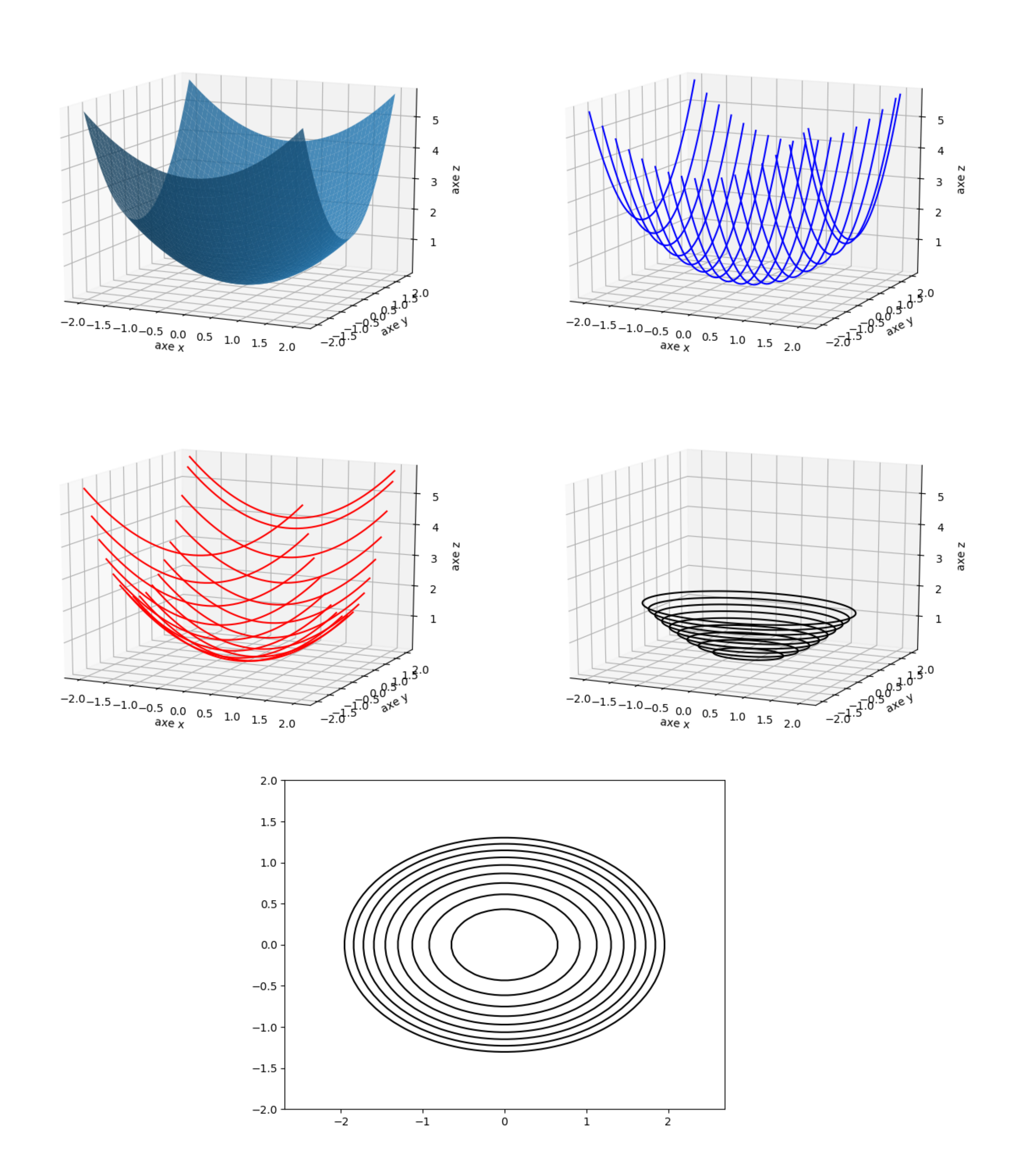
Ce sont des exemples à bien comprendre car ils seront importants pour la suite du cours.

Exemple.

$$f(x,y) = \frac{x^2}{a^2} + \frac{y^2}{b^2} - 1$$

- Les tranches sont des paraboles.
- Les lignes de niveau sont des ellipses.
- Le graphe est donc un ***paraboloïde elliptique***.

Ci-dessous : (a) la surface, (b) les tranches avec x constant, (c) les tranches avec y constant, (d) les courbes de niveau, (e) les lignes de niveau dans le plan.

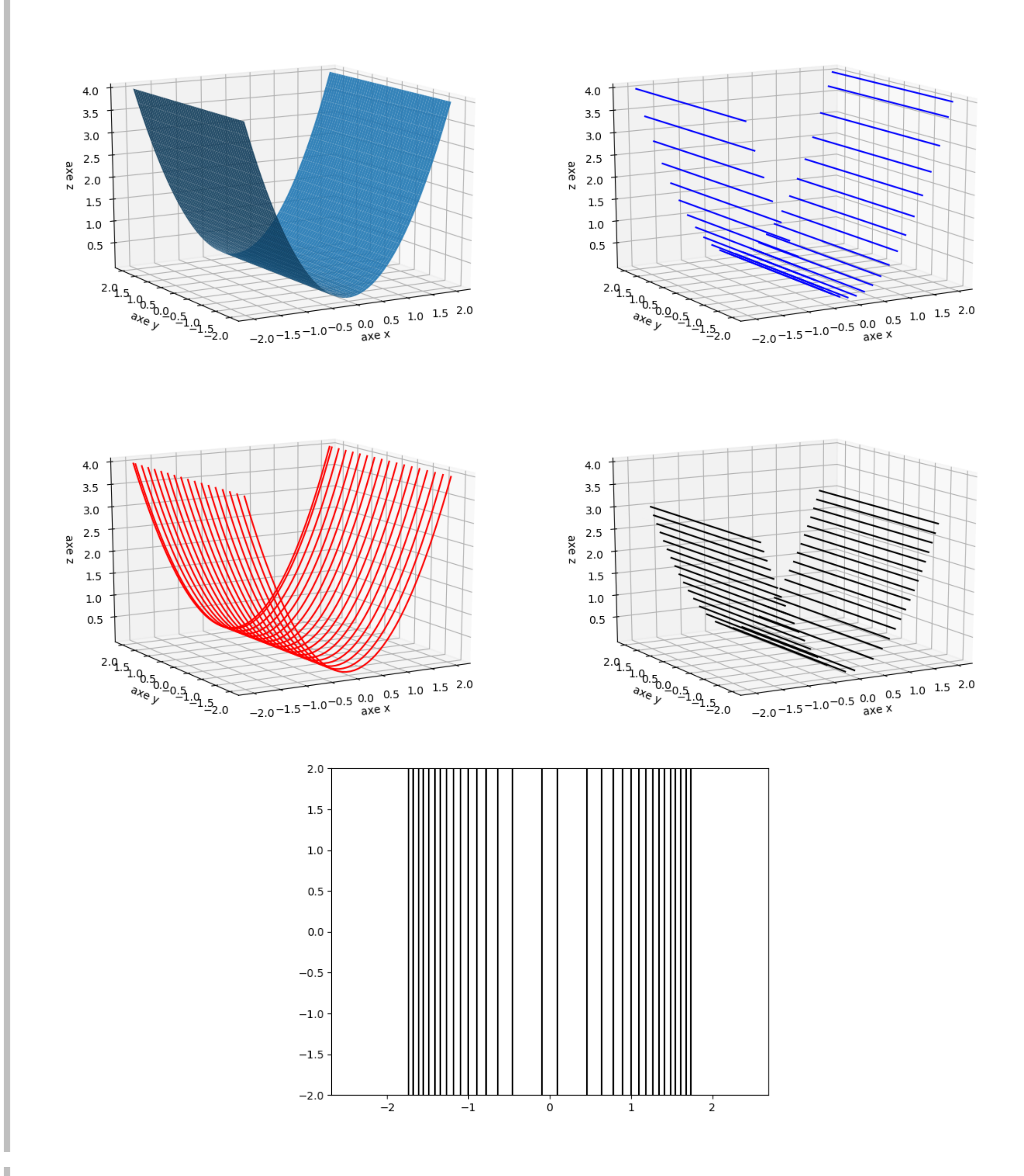


Exemple.

$$f(x,y) = x^2$$

- Les tranches obtenues en coupant selon des plans $y = y_0$ sont des paraboles. Dans l'autre direction, ce sont des droites.
- Les lignes de niveau sont des droites.
- Le graphe est donc un ***cylindre parabolique***.

Ci-dessous : (a) la surface, (b) les tranches avec x constant, (c) les tranches avec y constant, (d) les courbes de niveau, (e) les lignes de niveau dans le plan.

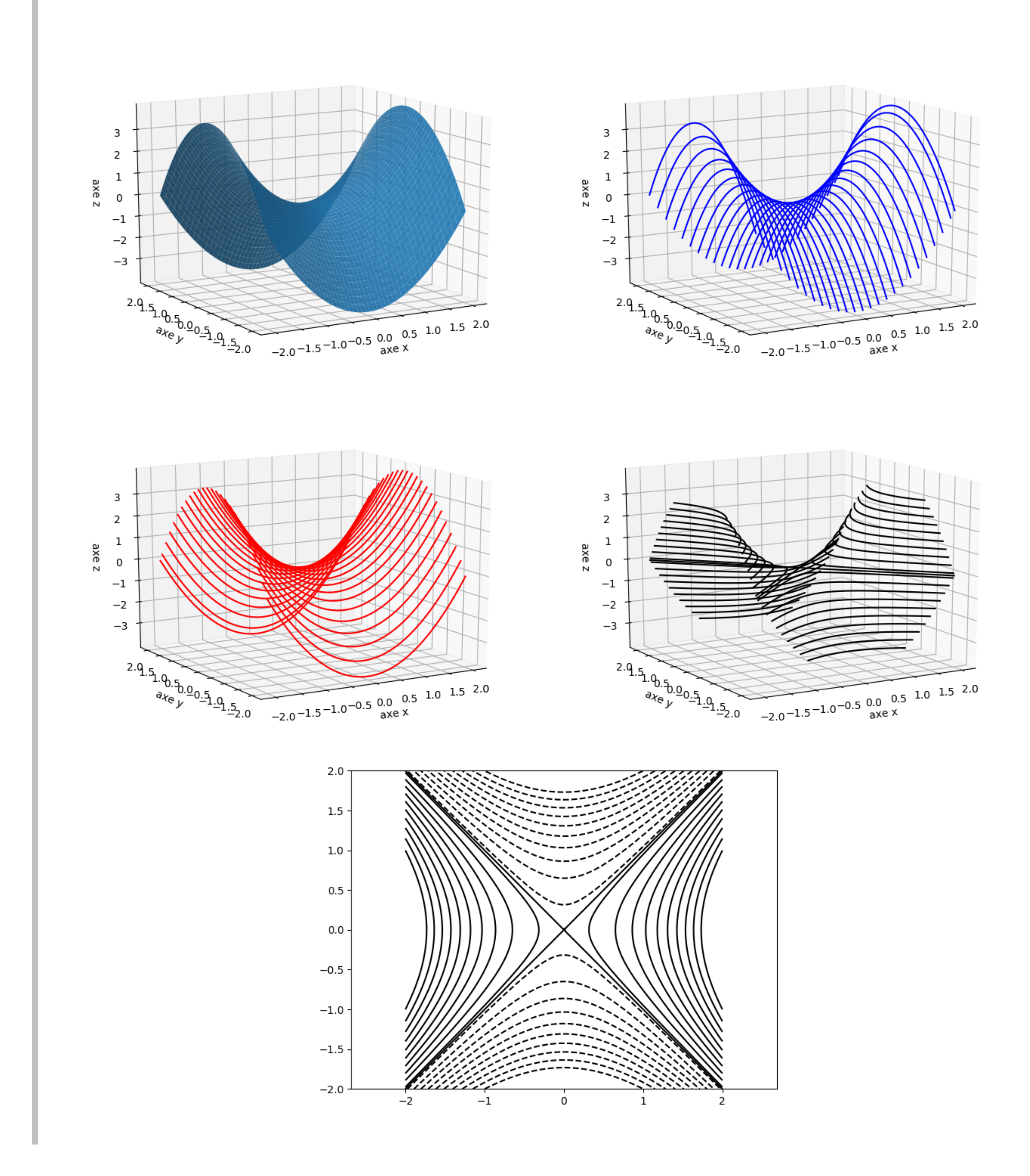


Exemple.

$$f(x,y) = x^2 - y^2$$

- Les tranches sont des paraboles, tournées vers le haut ou vers le bas selon la direction de la tranche.
- Les lignes de niveau sont des hyperboles.
- Le graphe est donc un ***paraboloïde hyperbolique*** que l'on appelle aussi la ***selle de cheval***.
- Un autre nom pour cette surface est un ***col*** (en référence à un col en montagne). En effet le point $(0,0,0)$, est le point de passage le moins haut pour passer d'un versant à l'autre de la montagne.

Ci-dessous : (a) la surface, (b) les tranches avec x constant, (c) les tranches avec y constant, (d) les courbes de niveau, (e) les lignes de niveau dans le plan (en pointillé les lignes de niveau négatif).

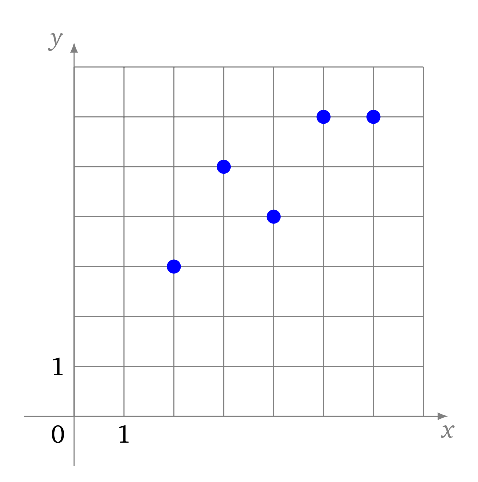


3.4. Régression linéaire

Exemple.
On se donne des points du plan, comme ci-dessous. On s'aperçoit qu'ils sont à peu près alignés et on souhaite trouver l'équation d'une droite :

$$y = ax + b$$

qui les approche au mieux.



Formalisons un peu le problème : on se donne N points $A_i(x_i, y_i)$, $i = 1, \ldots, N$. Pour une droite $\mathcal{D}$ d'équation $y = ax + b$, la distance entre A_i et la droite $\mathcal{D}$ est donnée par la formule :

$$d(A_i, \mathcal{D}) = \frac{|ax_i - y_i + b|}{\sqrt{1+a^2}}.$$

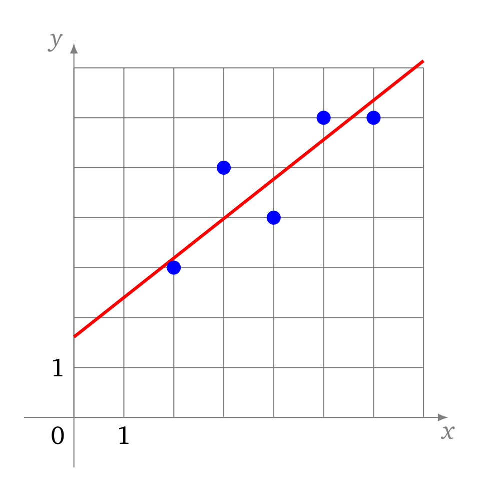
Pour se débarrasser des valeurs absolues et des racines carrées, on élève au carré et on décide que la droite qui approche au mieux tous les points A_i est la droite qui minimise la fonction

$$f(a, b) = \sum_{i=1}^{N} d(A_i, \mathcal{D})^2 = \frac{1}{1+a^2} \sum_{i=1}^{N} (ax_i - y_i + b)^2.$$

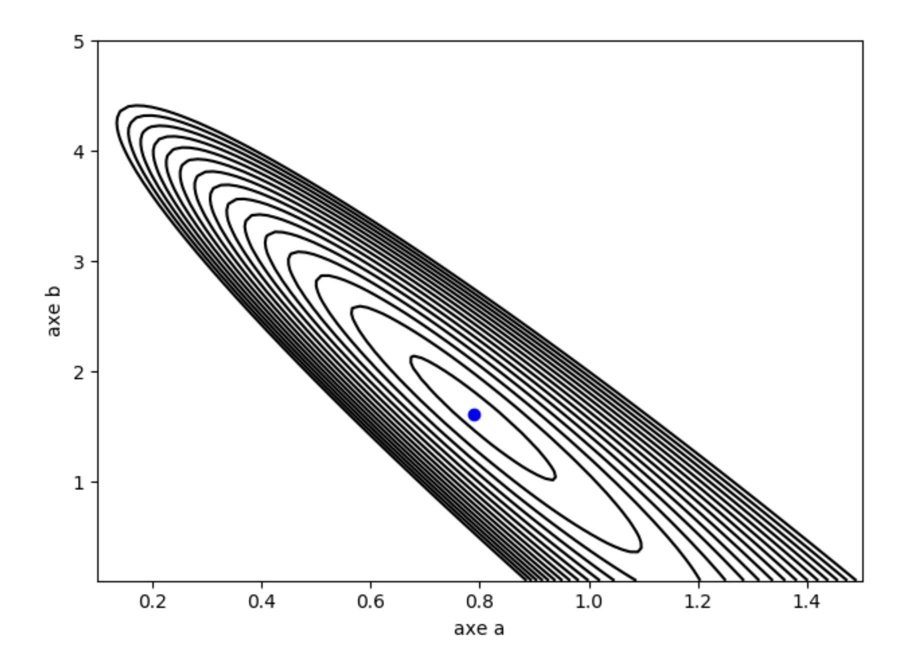
Pour les 5 points du dessin initial : $A_1(2,3)$, $A_2(3,5)$, $A_2(4,4)$, $A_4(5,6)$ et $A_5(6,6)$, il s'agit donc de trouver (a, b) qui minimise la fonction

$$f(a, b) = \frac{1}{1+a^2}\Big((2a-3+b)^2 + (3a-5+b)^2 + (4a-4+b)^2 + (5a-6+b)^2 + (6a-6+b)^2\Big).$$

On trace le graphe de f, les lignes de niveau de f, et on utilise les techniques à notre disposition pour trouver qu'un minimum global est réalisé en $(a, b) \simeq (0.8, 1.6)$, ce qui permet de tracer une droite $y = ax + b$ solution.



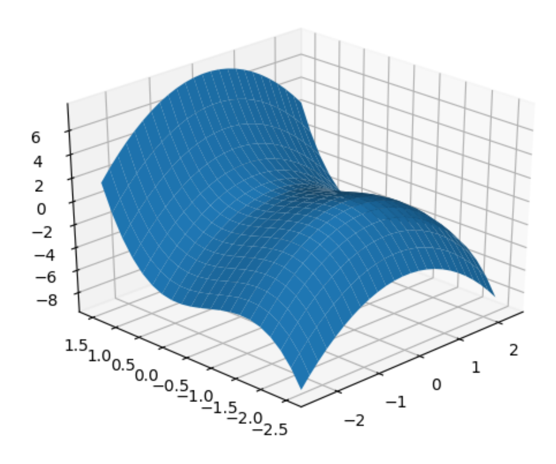
Lorsque l'on dessine les lignes de niveau, on s'aperçoit que le minimum (le point bleu) se trouve dans une région plate et allongée. Cela signifie que, bien que le minimum $(a_{\min}, b_{\min})$ soit unique, il existe beaucoup de (a, b) tels que $f(a, b)$ soit proche de la valeur minimale $f(a_{\min}, b_{\min})$. De plus, ces points (a, b) peuvent être assez éloignés de la solution $(a_{\min}, b_{\min})$ (par exemple tous les points de la zone ovale autour du point bleu). Ce qui signifie que beaucoup de droites très différentes approchent la solution optimale.


5
4
3
2
1
axe b
0.2
0.4
0.6
0.8
1.0
1.2
1.4
axe a

Chapitre 4

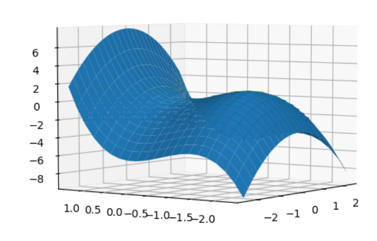
Python : numpy et matplotlib avec deux variables

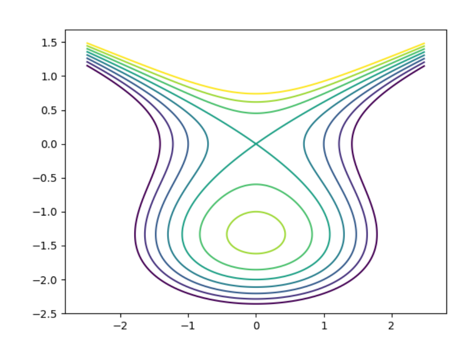
Vidéo ■ partie 4.1. Numpy
Vidéo ■ partie 4.2. Matplotlib



Le but de ce chapitre est d'approfondir notre connaissance de numpy *et* matplotlib *en passant à la dimension 2. Nous allons introduire les tableaux à double entrée qui sont comme des matrices et visualiser les fonctions de deux variables.*

Voici le graphe de la fonction $f(x,y) = y^3 + 2y^2 - x^2$ *(ci-dessus et ci-dessous à gauche, dessinés sous deux points de vue différents) ainsi que ses lignes de niveau dans le plan (ci-dessous à droite).*





1. Numpy (deux dimensions)

Nous avons vu comment définir un vecteur (un tableau à une dimension), comme par exemple `[1 2 3 4]`. Nous allons maintenant étudier les tableaux à deux dimensions.

1.1. Tableau

Un tableau à deux dimensions est comme une matrice (ou un tableau à double entrée).

- **Définition.** Un tableau se définit comme une suite de lignes

```
A = np.array([[1, 2, 3], [4, 5, 6]])
```

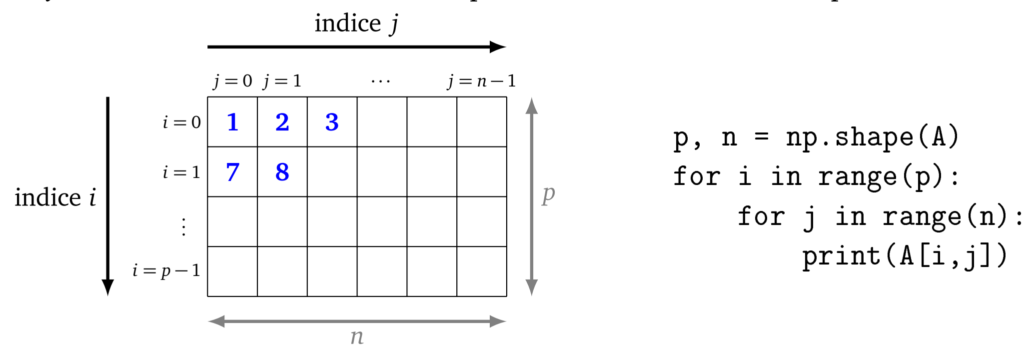
  (en ayant au préalable importé et renommé le module *numpy* en `np`) et s'affiche ainsi :

```
[[1 2 3]
 [4 5 6]]
```

  C'est un tableau à deux lignes et trois colonnes qui correspond à la matrice :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}.$$

- **Taille.** On récupère la taille du tableau par la fonction `shape()`. Par exemple `np.shape(A)` renvoie ici $(2,3)$, pour 2 lignes et 3 colonnes.
- **Parcourir les éléments.** On accède aux éléments par des instructions du type `A[i,j]` où i est le numéro de ligne et j celui de la colonne. Voici le code pour afficher les éléments un par un.



Tableau

- **Fonctions.** Les fonctions s'appliquent élément par élément. Par exemple `np.sqrt(A)` renvoie un tableau ayant la même forme, chaque élément étant la racine carrée de l'élément initial.

```
[[1.         1.41421356 1.73205081]
 [2.         2.23606798 2.44948974]]
```

- **Définition (suite).** `np.zeros((p,n))` renvoie un tableau de p lignes et n colonnes rempli de 0. La fonction `np.ones()` fonctionne sur le même principe.

1.2. Conversion tableau-vecteur

Il est facile et souvent utile de passer d'un tableau à un vecteur et réciproquement.

- **Tableau vers vecteur.** On obtient tous les éléments d'un tableau regroupés dans un vecteur par la commande d'aplatissement `flatten()`. Par exemple si

```
A = np.array([[1, 2, 3], [4, 5, 6]])
```

  alors la commande `X = A.flatten()` renvoie le vecteur X :

```
[1 2 3 4 5 6]
```

- **Vecteur vers tableau.** L'opération inverse se fait avec la fonction `reshape()` qui prend en entrée la nouvelle taille désirée. Par exemple : `X.reshape((2,3))` redonne exactement `A`. Par contre `X.reshape((3,2))` renvoie un tableau à 3 lignes et 2 colonnes :

```
[[1 2]
 [3 4]
 [5 6]]
```

1.3. Fonctions de deux variables

- **Évaluation sur des vecteurs.** Supposons que nous ayons défini une fonction *Python* de deux variables `f(x,y)`. Si `VX` et `VY` sont deux vecteurs de même taille, alors `f(VX,VY)` renvoie un vecteur composé des $f(x_i, y_i)$ pour x_i dans `VX` et y_i dans `VY`. Par exemple, si `VX` vaut `[1 2 3 4]` et `VY` vaut `[5 6 7 8]` alors `f(VX,VY)` est le vecteur de longueur 4 composé de $f(1,5)$, $f(2,6)$, $f(3,7)$, $f(4,8)$. Mais ceci n'est pas suffisant pour tracer des fonctions de deux variables.
- **Grille.** Pour dessiner le graphe d'une fonction $f(x, y)$, il faut calculer des valeurs $z = f(x, y)$ pour des (x, y) parcourant une grille. Voici comment définir simplement une grille à l'aide `meshgrid()`.

```
n = 5
VX = np.linspace(0, 2, n)
VY = np.linspace(0, 2, n)
X,Y = np.meshgrid(VX, VY)

def f(x,y):
    return x**2 + y**2

Z = f(X,Y)
```

- **Explications.**
  — `VX` est un découpage de l'axe des x en n valeurs.
  — `VY` est la même chose pour l'axe des y.
  — `X` et `Y` renvoyés par `meshgrid()` forment la grille. Ce sont des tableaux $n \times n$. Le premier représente les abscisses des points de la grille, le second les ordonnées. (Ce n'est pas très naturel mais c'est comme ça !)

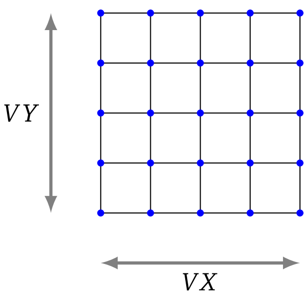


Tableau `X`

```
[[0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]
 [0.  0.5 1.  1.5 2. ]]
```

Tableau `Y`

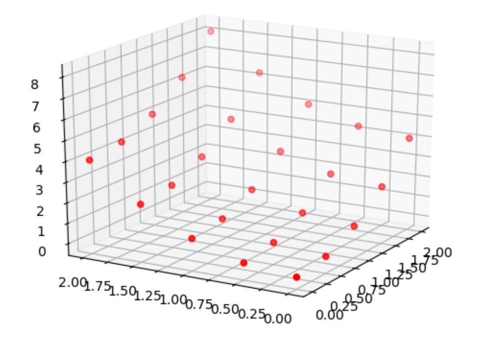
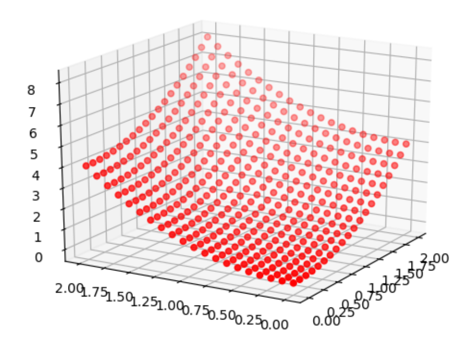
```
[[0.  0.  0.  0.  0. ]
 [0.5 0.5 0.5 0.5 0.5]
 [1.  1.  1.  1.  1. ]
 [1.5 1.5 1.5 1.5 1.5]
 [2.  2.  2.  2.  2. ]]
```

  — Enfin `Z = f(X,Y)` calcule les valeurs $z = f(x, y)$ pour tous les éléments de (x, y) de la grille et renvoie un tableau des valeurs.

Tableau Z

```
[[0.   0.25 1.   2.25 4.  ]
 [0.25 0.5  1.25 2.5  4.25]
 [1.   1.25 2.   3.25 5.  ]
 [2.25 2.5  3.25 4.5  6.25]
 [4.   4.25 5.   6.25 8.  ]]
```

- **Points.** Ainsi X, Y, Z forment une liste des n^2 points de l'espace dont on donne d'abord l'abscisse (dans X), puis l'ordonnée (dans Y), puis la hauteur (dans Z). Si on augmente la valeur de n, on commence à voir apparaître la surface d'équation $z = f(x, y)$. Ci-dessous le cas $n = 5$ à gauche et $n = 20$ à droite.

2. Un peu plus sur numpy

2.1. Ses propres fonctions

On peut définir ses propres fonctions. Dans le cas le plus simple, il n'y a rien de spécial à faire.

```
def ma_formule(x):
    return np.cos(x)**2 - np.sin(x)**2
```

Alors `ma_formule(X)` renvoie le résultat, que X soit un nombre, un vecteur ou un tableau.
On peut aussi utiliser les fonctions « lambda » :

```
f = lambda n:n*(n+1)/2
```

à utiliser sous la forme usuelle `Y = f(X)`.

2.2. Ses propres fonctions (suite)

Par contre, la fonction suivante ne sait traiter directement ni un vecteur ni un tableau, à cause du test de positivité.

```
def valeur_absolue(x):
    if x >= 0:
        return x
    else:
        return -x
```

Un appel `valeur_absolue(x)` fonctionne lorsque x est un nombre, mais si X est un vecteur ou un tableau alors un appel `valeur_absolue(X)` renvoie une erreur. Il faut « vectoriser » la fonction par la commande :

```
vec_valeur_absolue = np.vectorize(valeur_absolue)
```

Maintenant `vec_valeur_absolue(X)` fonctionne pour les nombres, les vecteurs et les tableaux.

Remarque.
Il peut être utile de préciser que chaque composante de sortie doit être un nombre flottant :

```
vec_fonction = np.vectorize(fonction, otypes=[np.float64])
```

2.3. Le zéro et l'infini

Le module *numpy* gère bien les problèmes rencontrés lors des calculs. Prenons l'exemple du vecteur X suivant :

```
[-1 0 1 2 3]
```

- La commande `1/X` renvoie le vecteur :
  ```
  [-1. inf 1. 0.5 0.33333333]
  ```
  La commande émet un avertissement (mais n'interrompt pas le programme). Comme valeur de 1/0 elle renvoie `inf` pour l'infini ∞.
- La commande `Y = np.log(X)` donne un vecteur Y :
  ```
  [ nan -inf 0. 0.69314718 1.09861229]
  ```
  où « `nan` » est une abréviation de *Not A Number* car le logarithme n'est pas défini pour des valeurs négatives et `-inf` représente $-\infty$ (qui est la limite en 0 du logarithme).
- La commande `Z = np.exp(Y)` donne `[nan 0. 1. 2. 3.]`. Ce qui est presque le vecteur X de départ et cohérent avec la formule $\exp(\ln(x)) = x$ pour $x > 0$.

2.4. Utilisation comme une liste

Soit le vecteur X défini par la commande :

```
X = np.linspace(0,10,num=100)
```

Pour récupérer une partie du vecteur, la syntaxe est la même que pour les listes *Python*.

- Élément de rang 50 : `X[50]`.
- Dernier élément : `X[-1]`.
- Éléments de rang 10 à 19 : `X[10:20]`.
- Éléments du début au rang 9 : `X[:10]`.
- Éléments du rang 90 à la fin : `X[90:]`.

Voici des fonctionnalités moins utiles.

- Ajouter un élément à un vecteur avec `append()`. Par exemple si `X = np.arange(0,5,0.5)` alors la commande `Y = np.append(X,8.5)` construit un nouveau vecteur qui se termine par l'élément 8.5.
- Revenir à une liste. Utiliser la conversion `list(X)` pour obtenir une liste *Python* à partir d'un vecteur *numpy*.

3. Matplotlib : deux variables

3.1. Graphes

On calcule d'abord une grille (X, Y) de points et les valeurs Z de la fonction sur cette grille. Ensuite le tracé du graphe se fait grâce à l'instruction `plot_surface(X, Y, Z)`.

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

n = 5
VX = np.linspace(-2.0, 2.0, n)
VY = np.linspace(-2.0, 2.0, n)
X,Y = np.meshgrid(VX, VY)

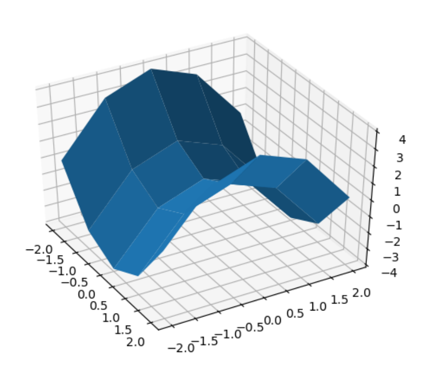
def f(x,y):
    return x**2-y**2

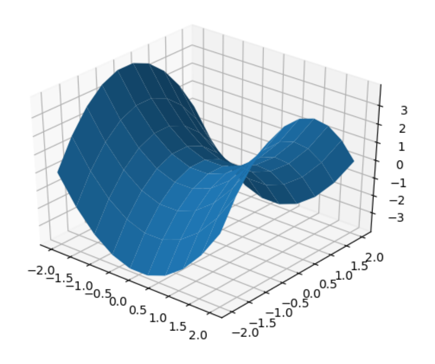
Z = f(X,Y)

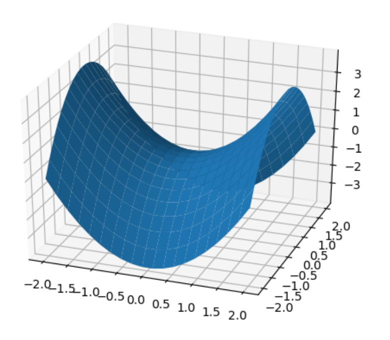
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.view_init(40, -30)

ax.plot_surface(X, Y, Z)

plt.show()
```

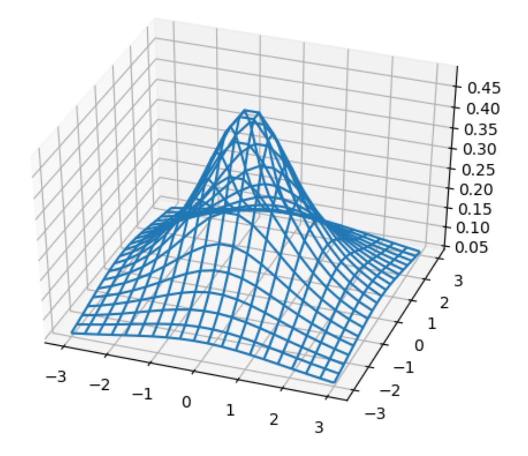






Si on augmente n, la grille est plus dense et la surface dessinée paraît plus lisse car il y a davantage de polygones. Les tracés de la « selle de cheval » ci-dessus sont effectués pour $n = 5$, $n = 10$, puis $n = 20$. L'affichage 3D permet de tourner la surface pour mieux l'appréhender. On peut aussi fixer le point de vue avec `view_init()`.

Il existe de multiples variantes et coloriages possibles.

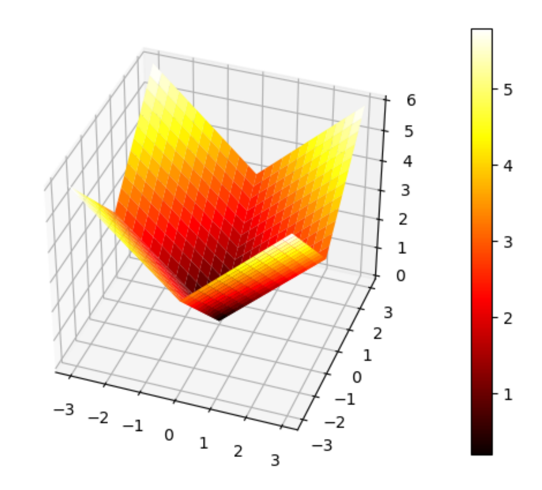


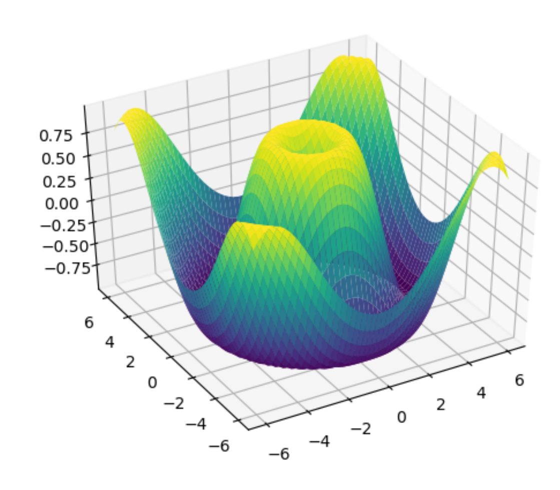
$$f(x,y) = \frac{1}{2+x^2+y^2}$$

`plot_wireframe(X, Y, Z)`

$f(x,y) = |x| + |y|$

Barre des couleurs



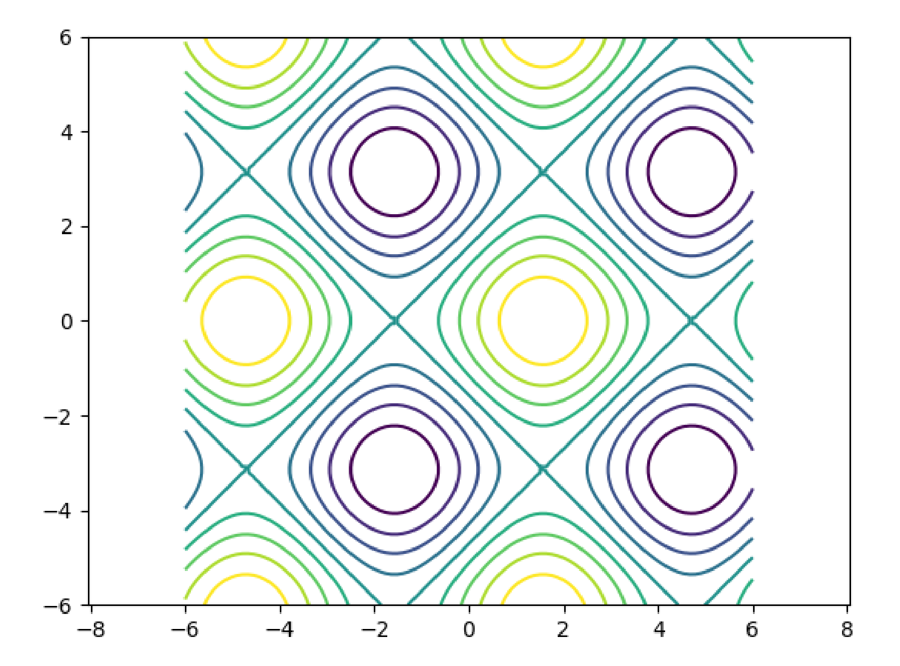


$$f(x,y) = \sin\left(\sqrt{x^2+y^2}\right)$$

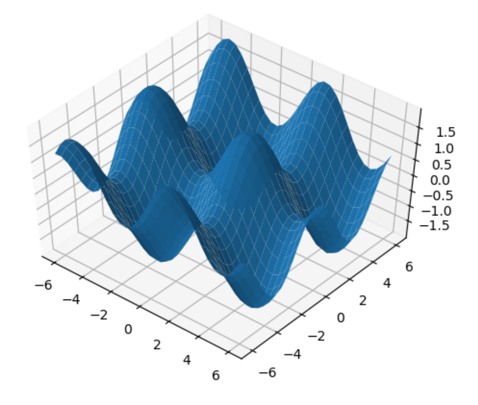
3.2. Lignes de niveau

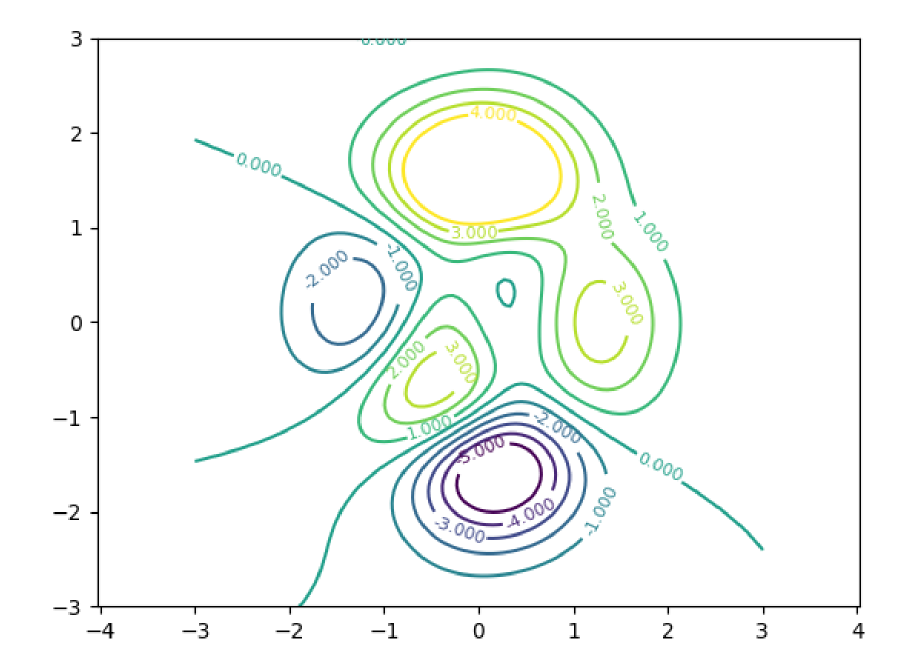
Il nous sera particulièrement utile de trouver les lignes de niveau d'une fonction f, en particulier pour trouver tous les (x,y) tels que $f(x,y) \geqslant 0$ par exemple. Le principe est similaire au tracé de la surface et s'effectue par la commande :

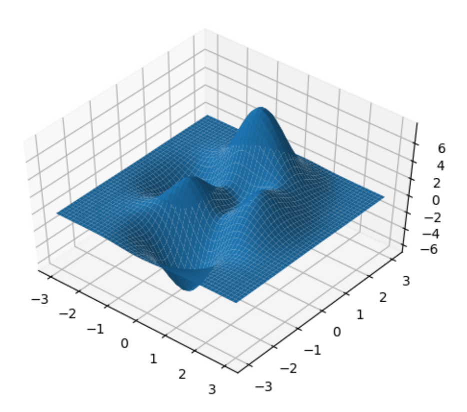
```
plt.contour(X, Y, Z)
```



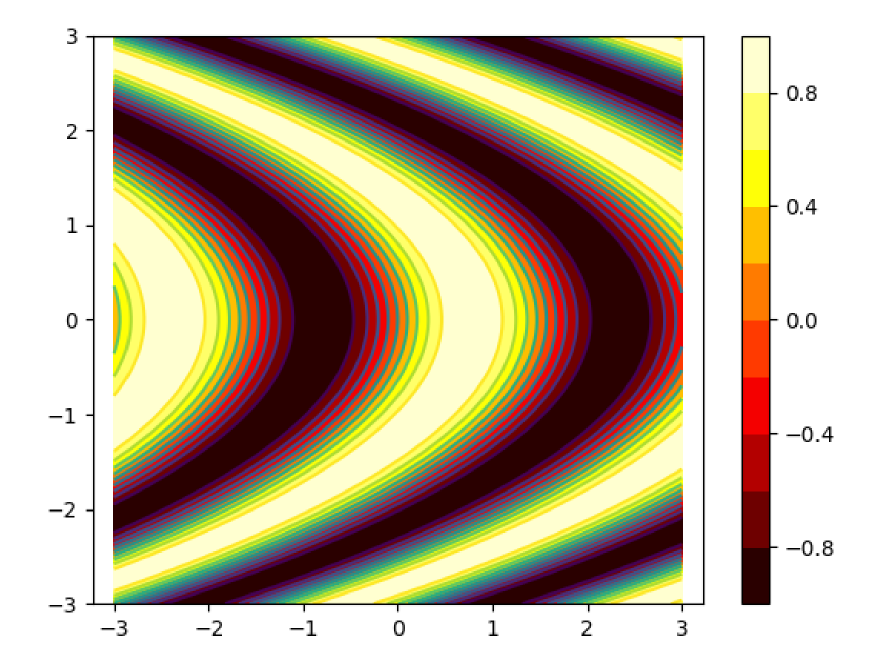
On peut préciser le nombre de niveaux tracés `plt.contour(X, Y, Z, nb_niveaux)` (ci-dessus). Le graphe en dimension 3 ci-contre.



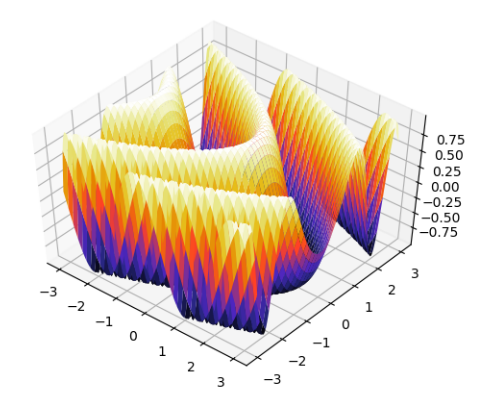




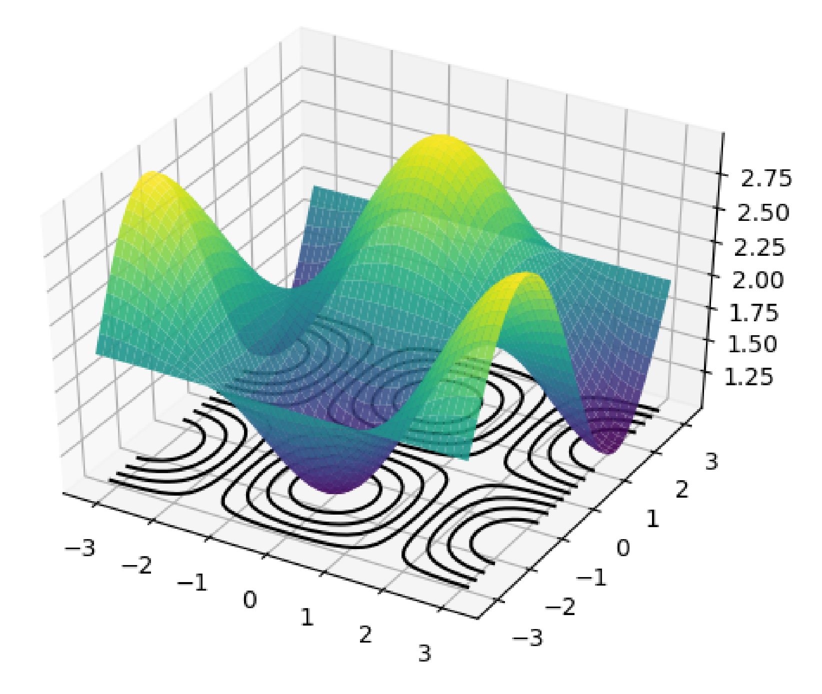
On peut spécifier les niveaux à afficher par `plt.contour(X, Y, Z, mes_niveaux)` où `mes_niveaux = np.arange(-5,5,1)` par exemple. On peut en profiter pour afficher la valeur du niveau, comme l'altitude sur une carte topographique (ci-dessus les lignes de niveau, ci-contre le graphe 3D).



`plt.contourf(X, Y, Z)` colorie le plan au lieu de tracer les lignes de niveau (ci-dessus les lignes de niveau, ci-contre le graphe 3D).



On peut même tracer les lignes de niveau sous la surface comme ci-dessous.


2.75
2.50
2.25
2.00
1.75
1.50
1.25
−3
−2
−1
0
1
2
3
3
2
1
0
−1
−2
−3

Chapitre 5

Réseau de neurones

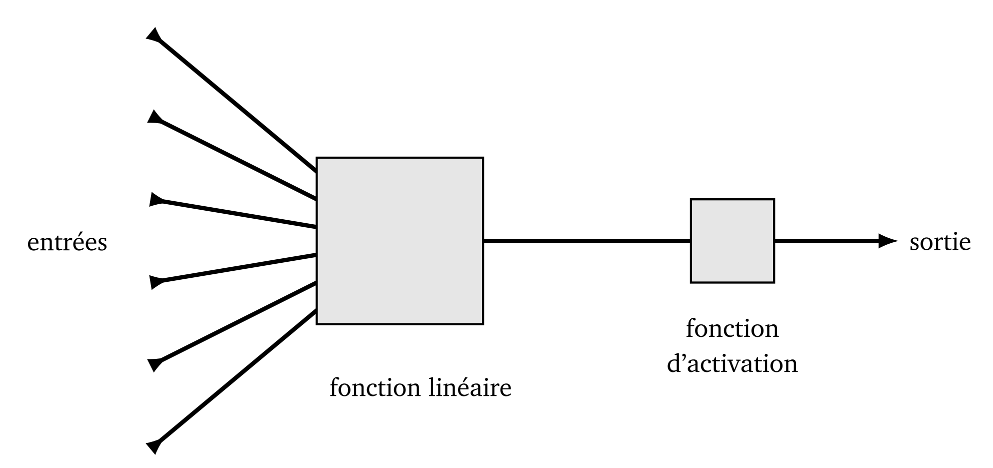
Vidéo ■ partie 5.1. Un neurone
Vidéo ■ partie 5.2. Théorie avec un neurone
Vidéo ■ partie 5.3. Plusieurs neurones
Vidéo ■ partie 5.4. Théorie avec plusieurs neurones
Vidéo ■ partie 5.5. Théorème d'approximation universelle

Le cerveau humain est composé de plus de 80 milliards de neurones. Chaque neurone reçoit des signaux électriques d'autres neurones et réagit en envoyant un nouveau signal à ses neurones voisins. Nous allons construire des réseaux de neurones artificiels. Dans ce chapitre, nous ne chercherons pas à expliciter une manière de déterminer dynamiquement les paramètres du réseau de neurones, ceux-ci seront fixés ou bien calculés à la main.

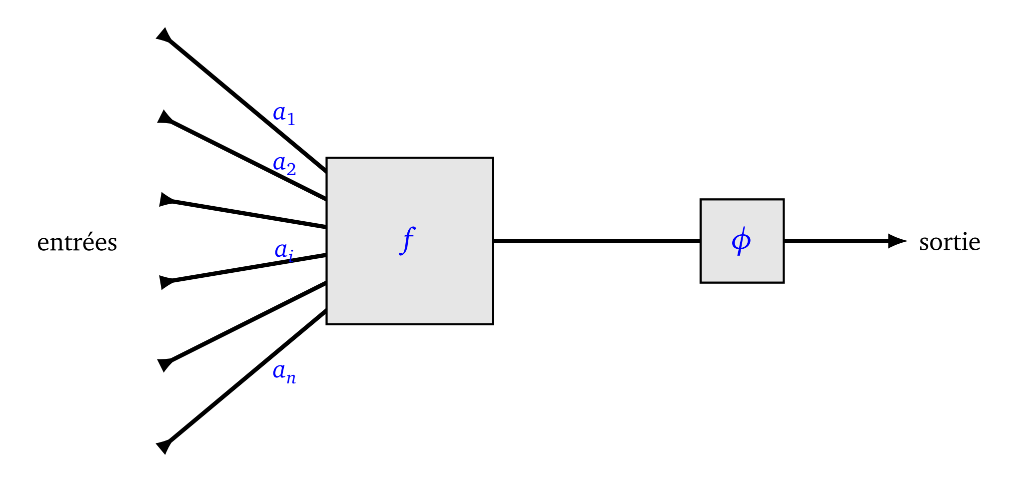
1. Perceptron

1.1. Perceptron linéaire

Le principe du perceptron linéaire est de prendre des valeurs en entrées, de faire un calcul simple et de renvoyer une valeur en sortie. Les calculs dépendent de paramètres propres à chaque perceptron.



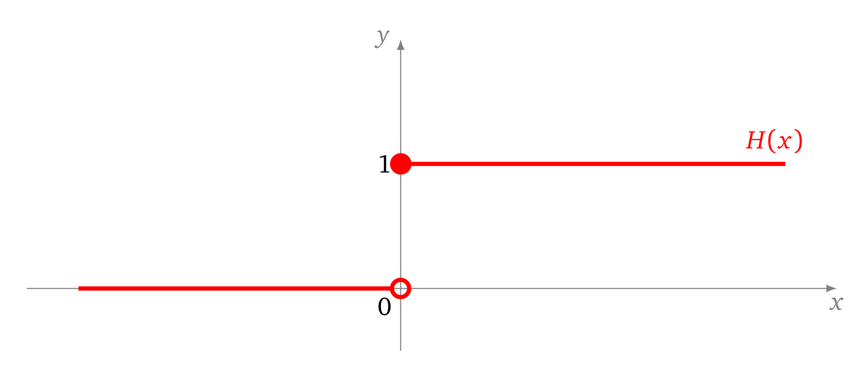
Le calcul effectué par un perceptron se décompose en deux phases : un calcul par une fonction linéaire f, suivi d'une fonction d'activation ϕ.



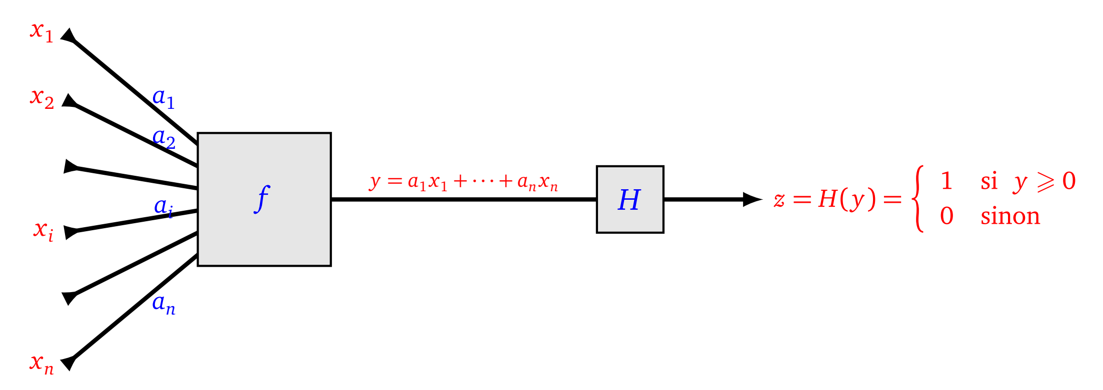
Détaillons chaque phase.

- **Partie linéaire.** Le perceptron est d'abord muni de ***poids*** $a_1, \ldots, a_n$ qui déterminent une fonction linéaire
$$f(x_1, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n.$$
- **Fonction d'activation.** La valeur renvoyée par la fonction linéaire f est ensuite composée par une fonction d'activation ϕ.
- **Sortie.** La valeur de sortie est donc $\phi(a_1 x_1 + a_2 x_2 + \cdots + a_n x_n)$.

Dans ce chapitre, la fonction d'activation sera (presque) toujours la fonction marche de Heaviside :
$$\begin{cases} H(x) = 1 & \text{si } x \geqslant 0, \\ H(x) = 0 & \text{si } x < 0. \end{cases}$$

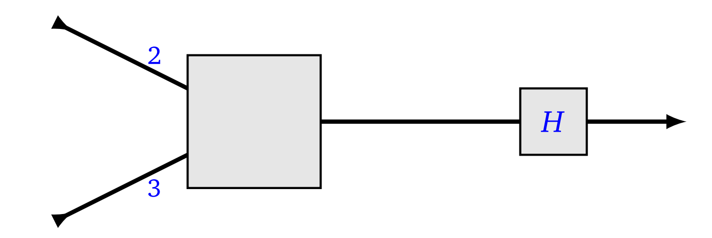


Voici ce que fait un perceptron linéaire de poids $a_1, \ldots, a_n$ et de fonction d'activation la fonction marche de Heaviside :

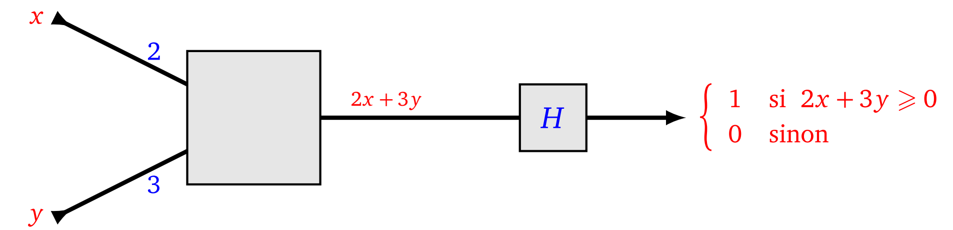


On peut donc définir ce qu'est un perceptron. Un ***perceptron linéaire*** à n variables et de fonction d'activation la fonction marche de Heaviside est la donnée de n coefficients réels $a_1, \ldots, a_n$ auxquels est associée la fonction $F : \mathbb{R} \to \mathbb{R}$ définie par $F = H \circ f$, c'est-à-dire :
$$\begin{cases} F(x_1, \ldots, x_n) = 1 & \text{si } a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \geqslant 0, \\ F(x_1, \ldots, x_n) = 0 & \text{sinon.} \end{cases}$$

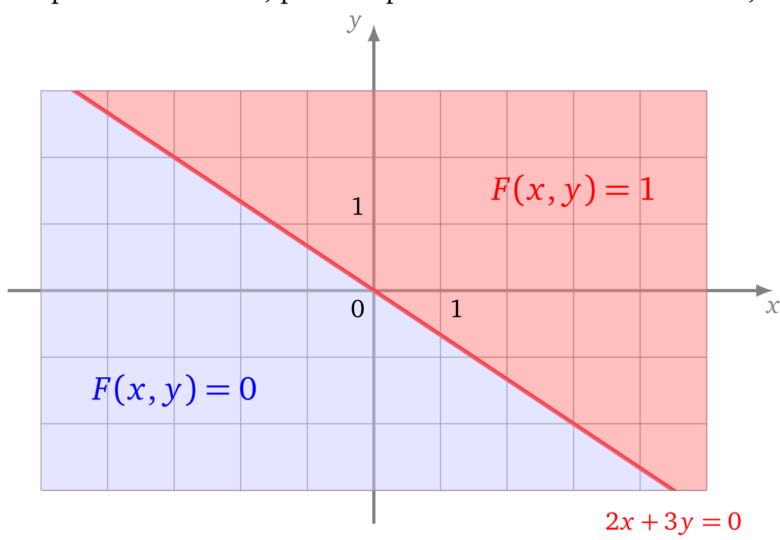
Exemple.
Voici un perceptron à deux entrées. Il est défini par les poids $a = 2$ et $b = 3$.



- **Formule.**
  Notons x et y les deux réels en entrée. La fonction linéaire f est donc
  $$f(x, y) = 2x + 3y.$$
  La valeur en sortie est donc :
  $$\begin{cases} F(x,y) = 1 & \text{si } 2x+3y \geqslant 0 \\ F(x,y) = 0 & \text{sinon.} \end{cases}$$

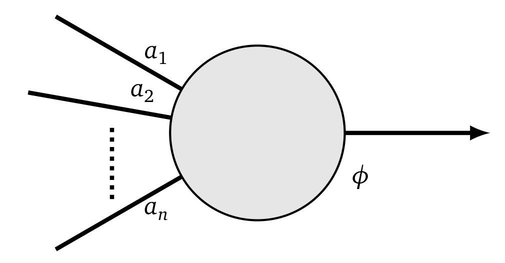


- **Évaluation.** Utilisons ce perceptron comme une fonction. Que renvoie le perceptron pour la valeur d'entrée $(x, y) = (4, -1)$? On calcule $f(x, y) = 2x + 3y = 5$. Comme $f(x, y) \geqslant 0$, alors la valeur de sortie est donc $F(x, y) = 1$.
  Recommençons avec $(x, y) = (-3, 1)$. Cette fois $f(x, y) = -3 < 0$ donc $F(x, y) = 0$.
  L'entrée $(x, y) = (6, -4)$ est « à la limite » car $f(x, y) = 0$ (0 est l'abscisse critique pour la fonction marche de Heaviside). On a $F(x, y) = 1$.
- **Valeurs de la fonction.**
  La fonction F prend seulement deux valeurs : 0 ou 1. La frontière correspond aux points (x, y) tels que $f(x, y) = 0$, c'est-à-dire à la droite $2x + 3y = 0$. Pour les points au-dessus de la droite (ou sur la droite) la fonction F prend la valeur 1 ; pour les points en-dessous de la droite, la fonction F vaut 0.

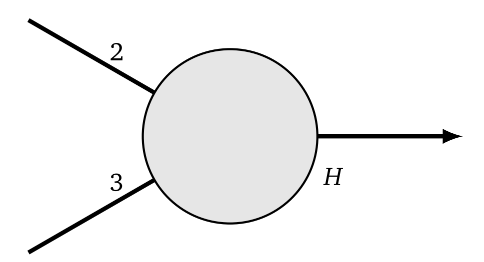


Notation.

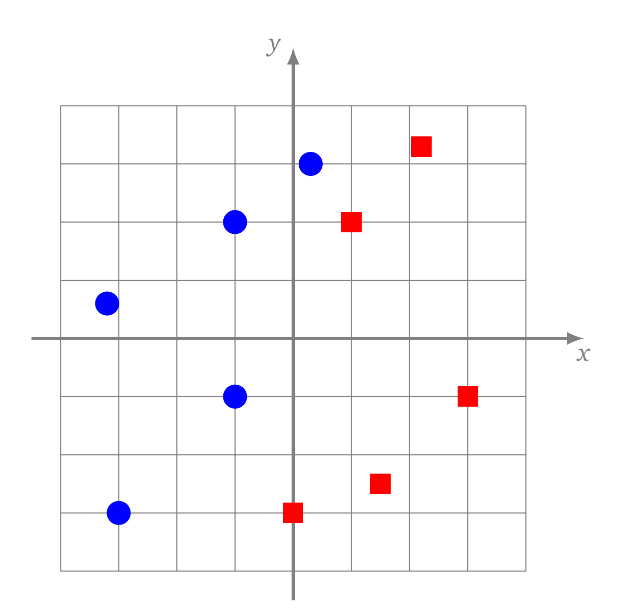
Nous représentons un perceptron par une forme plus condensée : sous la forme d'un ***neurone***, avec des poids sur les arêtes d'entrées. Nous précisons en indice la fonction d'activation utilisée ϕ. Si le contexte est clair cette mention est omise.



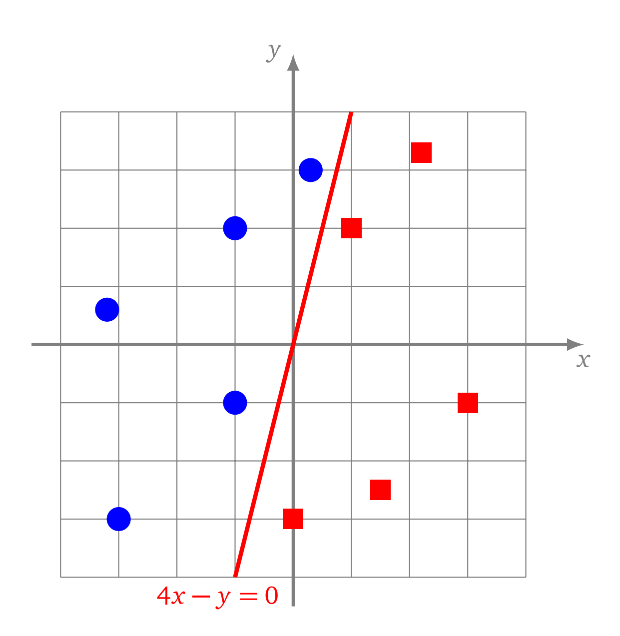
Voici le neurone à deux variables de l'exemple précédent.



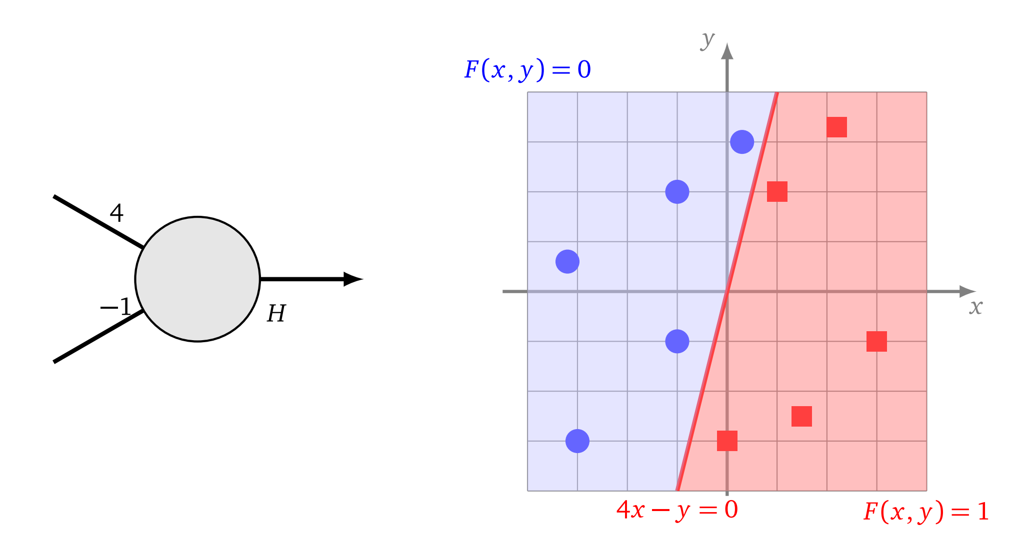
Exemple.
Voici deux catégories de points : des ronds bleus et des carrés rouges. Comment trouver un perceptron qui les sépare ?



Il s'agit donc de trouver les deux poids a et b d'un perceptron, dont la fonction associée F vérifie $F(x, y) = 1$ pour les coordonnées des carrés et $F(x, y) = 0$ pour les ronds.



Trouvons une droite qui les sépare. Par exemple, la droite d'équation $4x - y = 0$ sépare les ronds des carrés. On définit donc le neurone avec les poids $a = 4$ et $b = -1$. Si (x, y) sont les coordonnées d'un carré alors on a bien $F(x, y) = 1$ et pour un rond $F(x, y) = 0$.



1.2. Biais – Perceptron affine

Pour l'instant notre perceptron à deux entrées sépare le plan en deux parties selon une droite passant par l'origine. Nous allons plus loin avec le ***perceptron affine***.

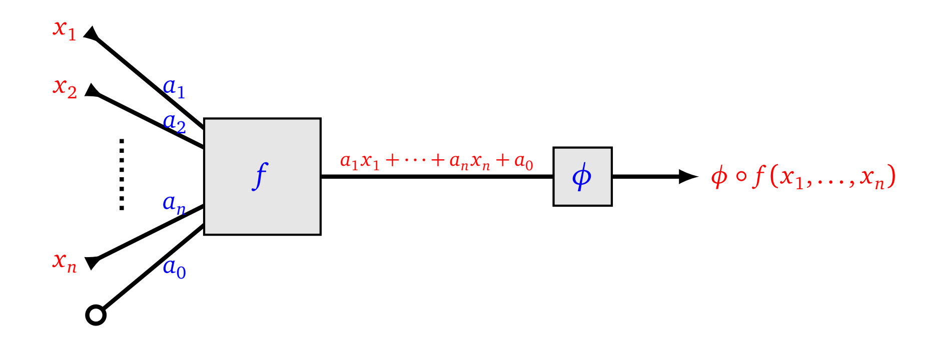
On modifie la définition du perceptron, sans changer le nombre d'entrées, mais en ajoutant un poids supplémentaire. Dans le cas de n entrées il y a donc $n + 1$ ***poids*** :

- les ***coefficients*** $a_1, \ldots, a_n \in \mathbb{R}$,
- et le ***biais*** $a_0 \in \mathbb{R}$,

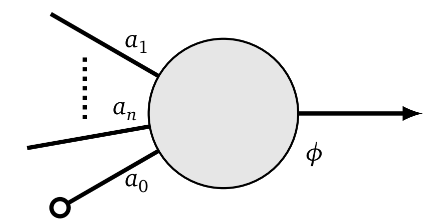
qui définissent la fonction affine :

$$f(x_1, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n + a_0.$$

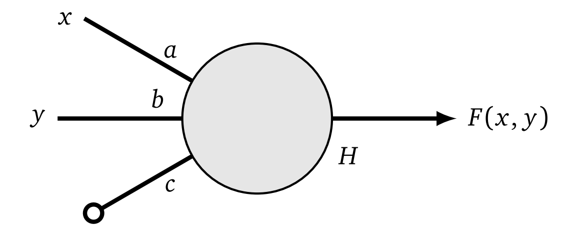
Comme auparavant, ce calcul est suivi de la fonction d'activation.



On représente le neurone avec une nouvelle arête, pondérée par le biais. L'arête est terminée par un rond pour signifier que cela ne correspond pas à une entrée.



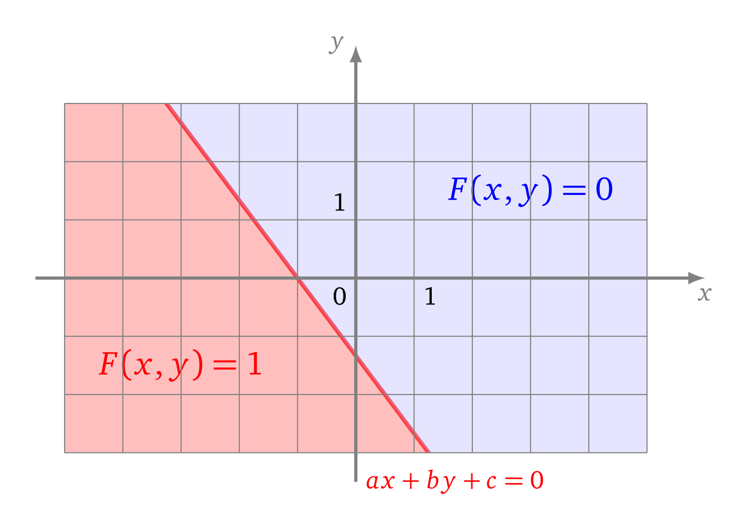
Dans le cas de deux entrées, les poids sont trois réels a, b (les coefficients) et c (le biais).

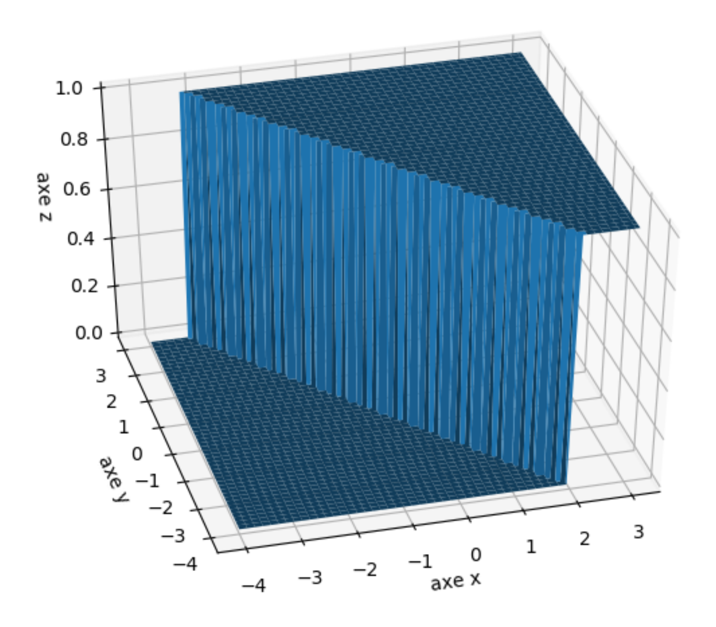


La fonction $f : \mathbb{R}^2 \to \mathbb{R}$ est la fonction affine $f(x, y) = ax + by + c$. Si la fonction d'activation est la fonction marche de Heaviside, alors le perceptron affine définit une fonction $F : \mathbb{R}^2 \to \mathbb{R}$ par la formule :

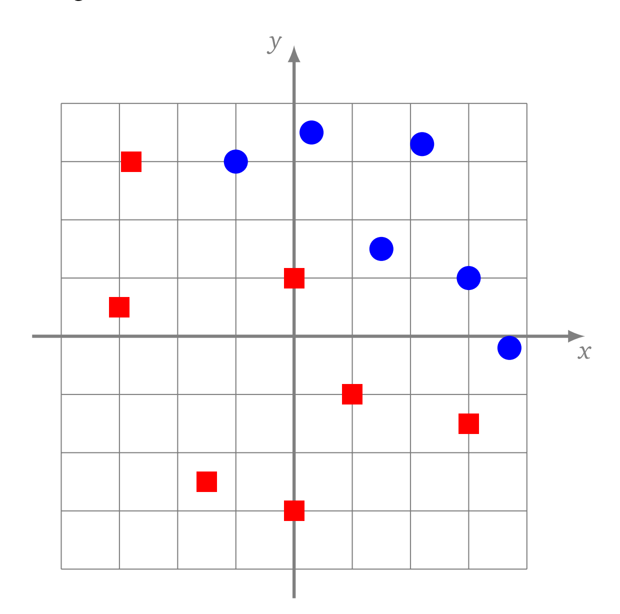
$$\begin{cases} F(x,y) = 1 & \text{si } ax + by + c \geqslant 0 \\ F(x,y) = 0 & \text{sinon.} \end{cases}$$

Un perceptron affine à deux entrées sépare donc le plan en deux parties, la frontière étant la droite d'équation $ax + by + c = 0$. D'un côté de cette droite la fonction F vaut 1, de l'autre elle vaut 0.





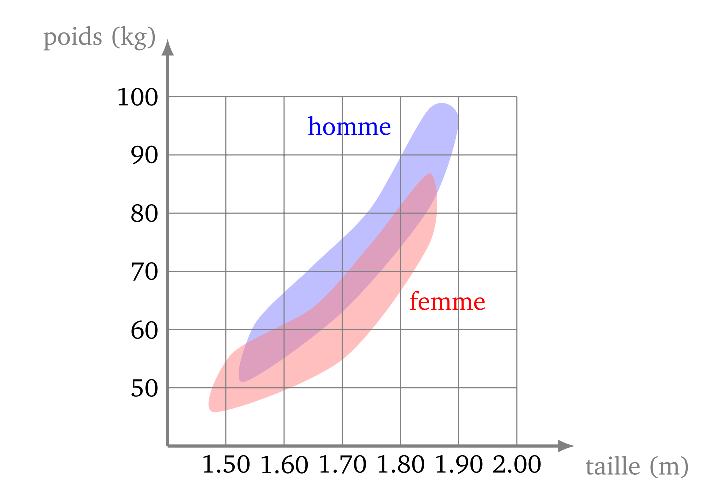
Exercice.
Trouver un perceptron qui distingue les carrés des ronds.



Il n'est pas toujours possible de répondre à une question seulement par « oui » ou « non ». Pour y remédier, on peut changer de fonction d'activation en utilisant par exemple la fonction sigmoïde σ. Dans ce cas, à chaque point du plan, on associe, non plus 0 ou 1, mais une valeur entre 0 et 1.
Ce nombre peut correspondre à un degré de certitude. Par exemple avec une question « Est-ce que cette photo est un chat ? », si la sortie vaut 0.8, cela signifie « c'est bien un chat avec 80% de certitude ». Si la sortie vaut 0.1 alors ce n'est probablement pas un chat.
Voici un neurone à deux entrées muni de la fonction d'activation sigmoïde définie par $\sigma(x) = \frac{1}{1+e^{-x}}$.

Fonction sigmoïde $y = \frac{1}{1+e^{-x}}$

Exemple.
Voici les données (fictives) de la répartition des hommes et des femmes selon leur taille et leur poids.



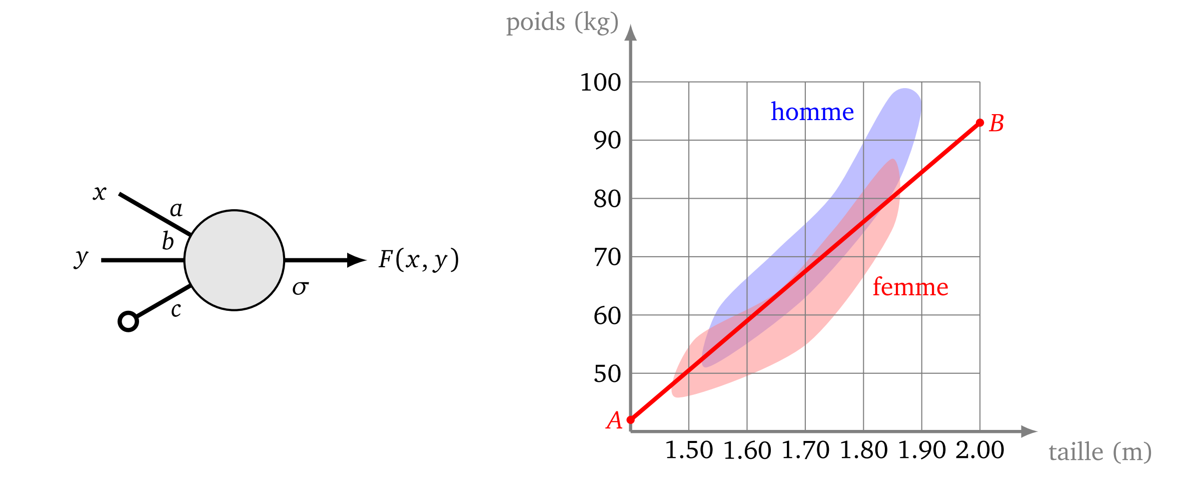
Problème : la taille et le poids d'une personne étant donnés, trouver un perceptron qui réponde à la question « Cette personne est-elle un homme ou une femme ? ».
Au vu de la superposition des zones, il n'est pas possible de répondre avec certitude. On construit donc un perceptron selon les idées suivantes :

- on trace une droite qui sépare au mieux les hommes des femmes, par exemple ici la droite qui passe par les points $A(1.40, 42)$ et $B(2.00, 93)$ d'équation approchée $y = 85x - 77$ où $(x, y) = (t, p)$ représente

la taille et le poids ;

- on choisit la fonction d'activation sigmoïde.

Ce qui nous permet de définir le perceptron suivant avec $a = 85$, $b = -1$ et $c = -77$.



Maintenant pour un couple donné (t, p), le perceptron associe une valeur $F(t, p) \in [0, 1]$. Si $F(t, p)$ est proche de 0 alors la personne est probablement un homme, si $F(t, p)$ est proche de 1 c'est probablement une femme.

Exemple : une personne mesurant 1.77 m et pesant 75 kg est-elle plutôt un homme ou une femme ? On calcule $f(1.77, 75) = -1.55$ où $f(x, y) = ax + by + c$. Puis on calcule $F(1.77, 75) = \sigma(-1.55) \simeq 0.17$. Selon notre modèle cette personne est probablement un homme (car la valeur de F est proche de 0).

Autre exemple : $(t, p) = (1.67, 64)$. On calcule $F(1.67, 64) \simeq 0.72$. Une personne mesurant 1.67m et pesant 64kg est probablement une femme (car la valeur de F est plus proche de 1 que de 0).

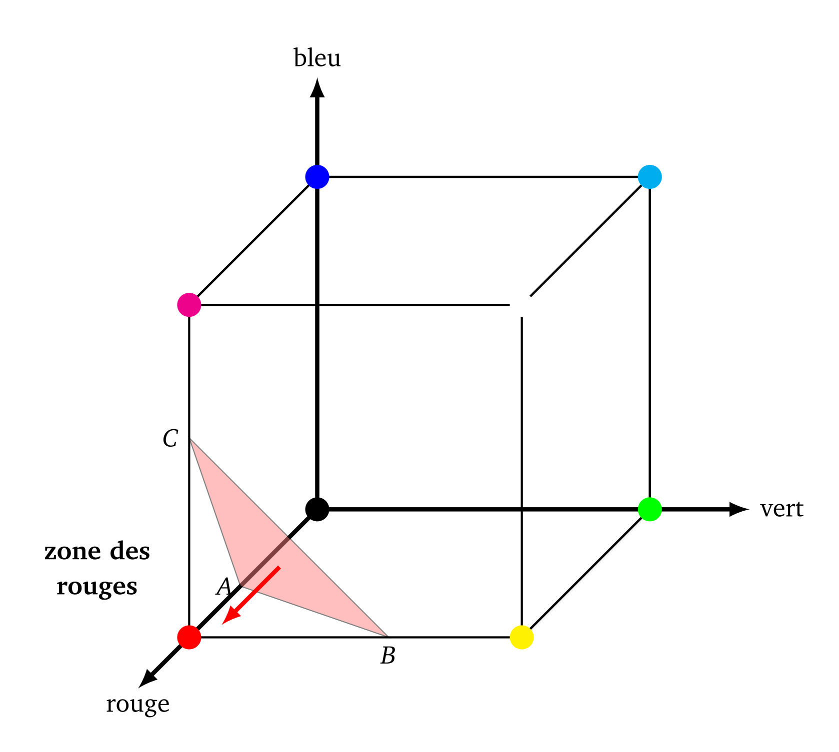
Malgré les données fictives, cet exemple met en évidence le problème de la superposition et de l'utilité d'avoir une sortie plus riche que « oui » ou « non ». On peut aussi discuter de la pertinence de la frontière, car la séparation par une droite ne semble pas la mieux adaptée. En fait, le poids varie en fonction du carré de la taille (les zones ont une forme de parabole).

Plus généralement un ***perceptron affine*** à n entrées est la donnée de $n + 1$ poids $a_0, a_1, \ldots, a_n \in \mathbb{R}$ et d'une fonction d'activation ϕ qui définissent une fonction $F : \mathbb{R}^n \to \mathbb{R}$ par la formule :

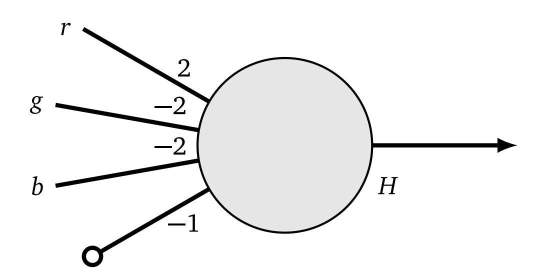
$$F(x_1, \ldots, x_n) = \phi(a_1 x_1 + \cdots + a_n x_n + a_0).$$

Exemple.

Dans le système de couleurs RGB (*red, green, blue*), une couleur est déterminée par trois réels $r, g, b \in [0, 1]$. Le « vrai » rouge est codé par $(1, 0, 0)$, mais bien sûr des couleurs proches sont aussi des nuances de rouge. On représente toutes les couleurs par un « cube de couleurs », chaque point du cube représente le code (r, g, b) d'une couleur.



On décide (un peu arbitrairement) que toute couleur située dans la zone du coin $(1,0,0)$ de la figure ci-dessus est une nuance de rouge.
Problème : trouver un perceptron qui répond à la question « Cette couleur est-elle une nuance de rouge ? ».
Solution. On considère que la zone est délimitée par le plan passant par les points $A(\frac{1}{2},0,0)$, $B(1,\frac{1}{2},0)$ et $C(1,0,\frac{1}{2})$ et les plans des faces du cube. Une équation de ce plan est $2x-2y-2z-1=0$ où en fait $(x,y,z)=(r,g,b)$. Le perceptron qui répond à la question est donc :

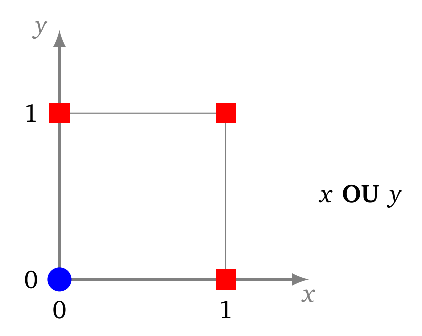


La fonction F associée, vérifie que $F(r,g,b)=1$ pour les points de la zone des rouges et $F(r,g,b)=0$ sinon.

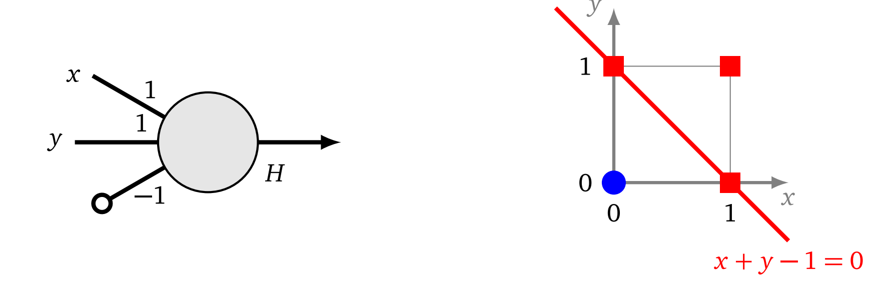
2. Théorie du perceptron

2.1. OU, ET, OU exclusif

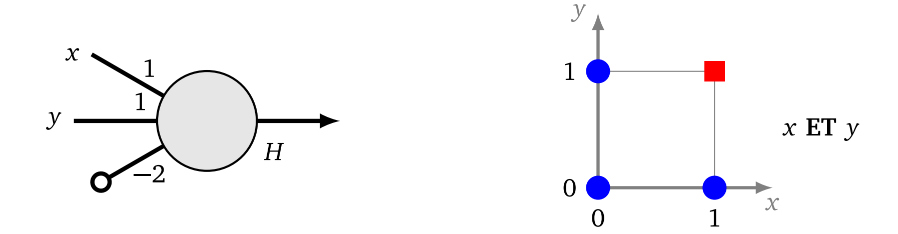
OU. Un *booléen* est une variable qui peut prendre soit la valeur « vrai », soit la valeur « faux ». Dans la pratique, on associe 1 à « vrai » et 0 à « faux ». À partir de deux booléens x et y, on peut associer un nouveau booléen « x OU y » qui est vrai lorsque x est vrai ou y est vrai.
Graphiquement, on représente toutes les configurations associées à « x OU y » par un diagramme. Un rond bleu en position (x,y) signifie que « x OU y » est faux (le résultat vaut 0), un carré rouge que « x OU y » est vrai (le résultat vaut 1).



Peut-on réaliser l'opération « x OU y » par un perceptron ? Oui ! Cela revient à séparer les ronds bleus des carrés rouges. C'est possible avec le perceptron de poids $a = 1$, $b = 1$, $c = -1$. Par exemple, si $x = 0$ et $y = 1$ alors, on calcule $1 \cdot 0 + 1 \cdot 1 - 1 = 0 \geqslant 0$. Composée avec la fonction marche de Heaviside, la fonction $F(x, y)$ définie par le perceptron renvoie dans ce cas 1 (« vrai »). Si $x = 0$ et $y = 0$, alors $1 \cdot 0 + 1 \cdot 0 - 1 = -1 < 0$ et $F(x, y) = 0$ (« faux »).

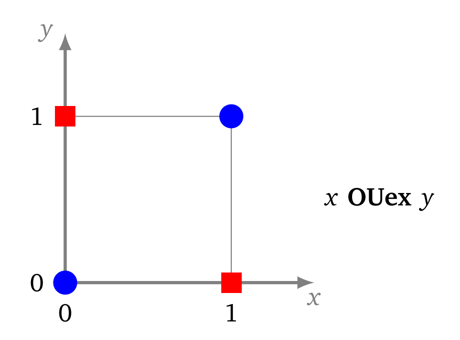


ET. On peut également réaliser l'opération « x ET y » (qui renvoie « vrai » uniquement si x et y sont vrais) en choisissant les poids $a = 1$, $b = 1$, $c = -2$.



Remarque.
D'un point de vue numérique, si on considère des valeurs réelles pour x et y, notre perceptron « ET » n'est pas numériquement très stable. Par exemple avec $x = 0.9$ et $y = 0.9$, on calcule $x + y - 2 = -0.2$ et la sortie est 0, mais comme x et y sont proches de 1, on aimerait que la sortie soit 1. Il suffit de changer un peu les paramètres : prenons $a = 1$, $b = 1$ et $c = -1.5$, alors $x + y - 1.5 = 0.3$ et cette fois la sortie est 1.

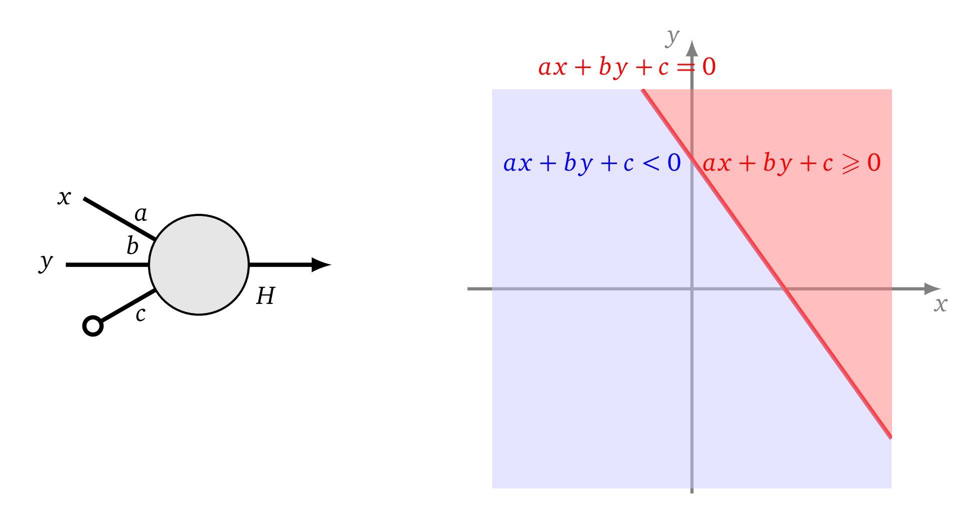
OU exclusif. Le ou exclusif, noté « x OUex y » est vrai lorsque x ou y est vrai, mais pas les deux en même temps. Le ou exclusif est-il réalisable par un perceptron ? Cette fois la réponse est non !



Proposition 1.
Il n'existe pas de perceptron (affine, à deux entrées et de fonction d'activation la fonction marche de Heaviside) qui réalise le « ou exclusif ».

Nous sommes convaincus géométriquement qu'il n'existe pas de droite qui sépare les ronds bleus des carrés rouges. Nous allons le prouver par un petit calcul.

Démonstration. Nous raisonnons par l'absurde en supposant qu'un tel perceptron existe. Nous cherchons une contradiction. Soit $a_1 = a, a_2 = b, a_0 = c$ les coefficients.



- Pour $(x, y) = (1, 0)$, on doit avoir $ax + by + c \geqslant 0$ (car après avoir composé avec la fonction d'activation le résultat doit être 1), donc
$$a + c \geqslant 0. \tag{1}$$
- De même pour $(x, y) = (0, 1)$, on doit avoir $ax + by + c \geqslant 0$, ce qui donne :
$$b + c \geqslant 0. \tag{2}$$
- Pour $(0, 0)$ et $(1, 1)$, on a $ax + by + c < 0$, ce qui implique :
$$c < 0, \tag{3}$$
$$a + b + c < 0. \tag{4}$$

Si on additionne les inégalités (1) et (2), on obtient $a + b + 2c \geqslant 0$. Par l'inégalité (3) on a $-c > 0$. Donc en ajoutant $-c$ à l'inégalité $a + b + 2c \geqslant 0$, on obtient $a + b + c > 0$. Ce qui contredit l'inégalité (4).
Conclusion : il ne peut exister trois réels a, b, c pour définir un perceptron réalisant le « ou exclusif ».

□

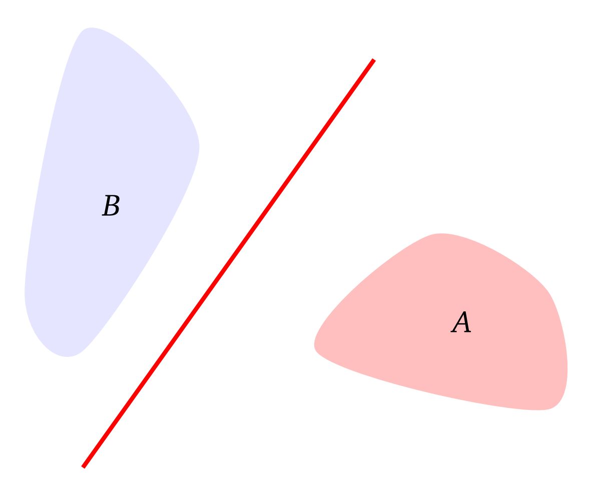
2.2. Séparation linéaire

Nous formalisons un peu les idées de la section précédente. Une ***droite*** du plan $\mathbb{R}^2$ est l'ensemble des points (x, y) vérifiant une équation du type $ax + by + c = 0$. Un ***plan*** de l'espace $\mathbb{R}^3$ est l'ensemble des points (x, y, z) vérifiant une équation du type $ax + by + cz + d = 0$. Nous généralisons cette notion en dimension plus grande.

Définition.
Un ***hyperplan (affine)*** de $\mathbb{R}^n$ est l'ensemble des points $(x_1, \dots, x_n)$ qui vérifient une équation du type :
$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n + a_0 = 0$$
où $a_1, \dots, a_n$ sont des réels (non tous nuls) et a_0 est aussi un réel.

Le but d'un perceptron affine est de séparer deux ensembles. Par exemple deux ensembles A et B du plan seront séparés par une droite, deux ensembles de l'espace sont séparés par un plan.

Définition.
Deux ensembles A et B sont ***linéairement séparables*** s'il existe un hyperplan qui sépare A de B. Plus précisément, s'il existe des réels $a_0, a_1, \dots, a_n$ tels que $a_1x_1+a_2x_2+\cdots+a_nx_n+a_0 \geqslant 0$ pour tout $(x_1, \dots, x_n) \in A$ et $a_1x_1 + a_2x_2 + \cdots + a_nx_n + a_0 < 0$ pour tout $(x_1, \dots, x_n) \in B$.



Nous résumons ce qui précède dans le résultat suivant.

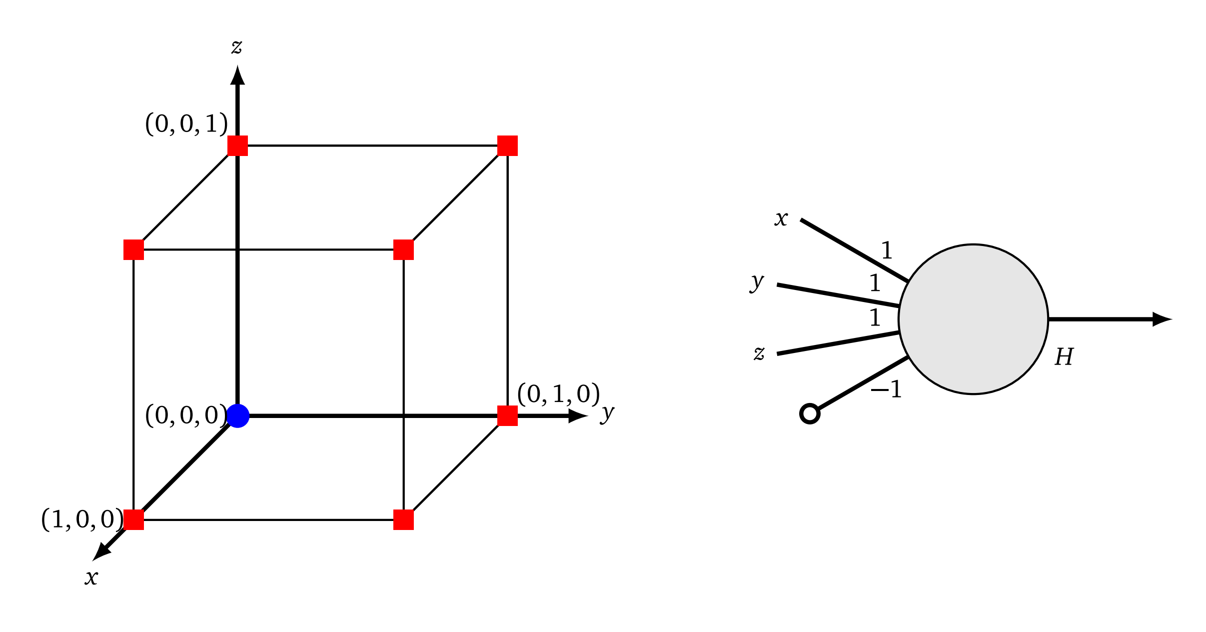
Proposition 2.
Deux ensembles A et B de $\mathbb{R}^n$ sont linéairement séparables si, et seulement si, il existe un perceptron affine dont la fonction F vaut 1 sur A et 0 sur B.

La preuve est simple : la fonction f définie par un perceptron (avant activation) est une fonction affine :

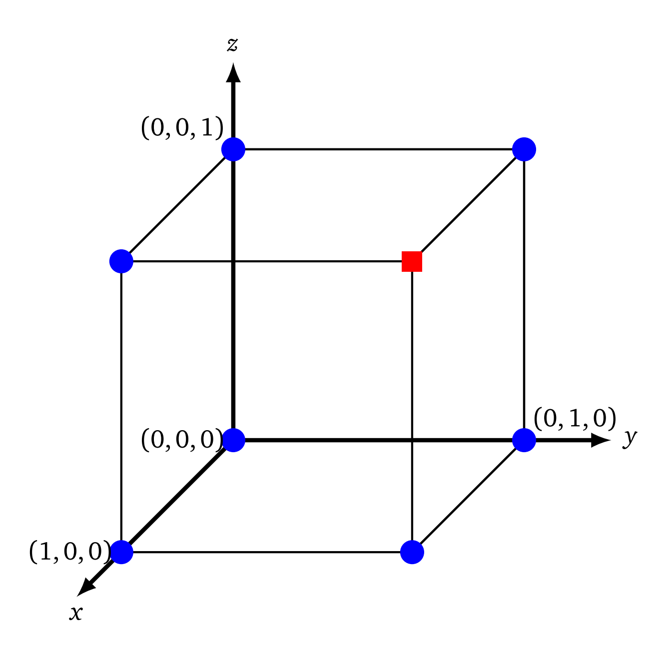
$$f(x_1, \dots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n + a_0.$$

Si $f(x_1, \dots, x_n) \geqslant 0$, alors après activation $F(x_1, \dots, x_n) = 1$, sinon $F(x_1, \dots, x_n) = 0$.

Exemple.
Réaliser « x OU y OU z » revient à séparer par un plan le rond bleu des carrés rouges du cube suivant. Trouver l'équation d'un de ces plans donne donc les poids du perceptron qui conviennent.



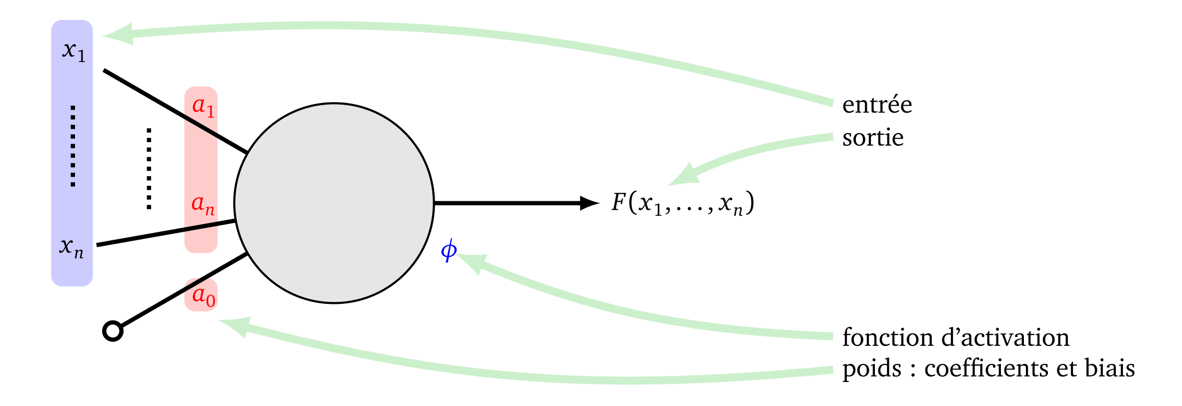
Exercice.
Trouver le perceptron qui réalise « x ET y ET z ».



2.3. Vocabulaire

Résumons le vocabulaire utilisé ci-dessus ainsi que les termes anglais correspondant :

- le **perceptron linéaire** ou **neurone artificiel**, a un biais qui vaut 0, à la différence du **perceptron affine** pour lequel le biais est un réel quelconque ;
- les **poids** (*weights*) ou **paramètres** sont les **coefficients** $a_1, \dots, a_n$ auxquels s'ajoute le **biais** (*bias*) (dans ce livre le biais est l'un des poids, ce qui n'est pas toujours le cas dans la littérature) ;
- un perceptron est la donnée des poids et d'une fonction d'activation ;
- chaque perceptron définit une fonction F qui est la composée d'une **fonction affine** f et d'une **fonction d'activation** ϕ ; la fonction d'activation la plus utilisée dans ce chapitre est la fonction marche de Heaviside (*step function*) ;
- on utilise un perceptron lors d'une **évaluation** : pour une **entrée** (*input*) $(x_1, \dots, x_n) \in \mathbb{R}^n$, on calcule la sortie (*output*) $F(x_1, \dots, x_n) \in \mathbb{R}$ (qui vaut 0 ou 1 dans le cas de la fonction marche de Heaviside).

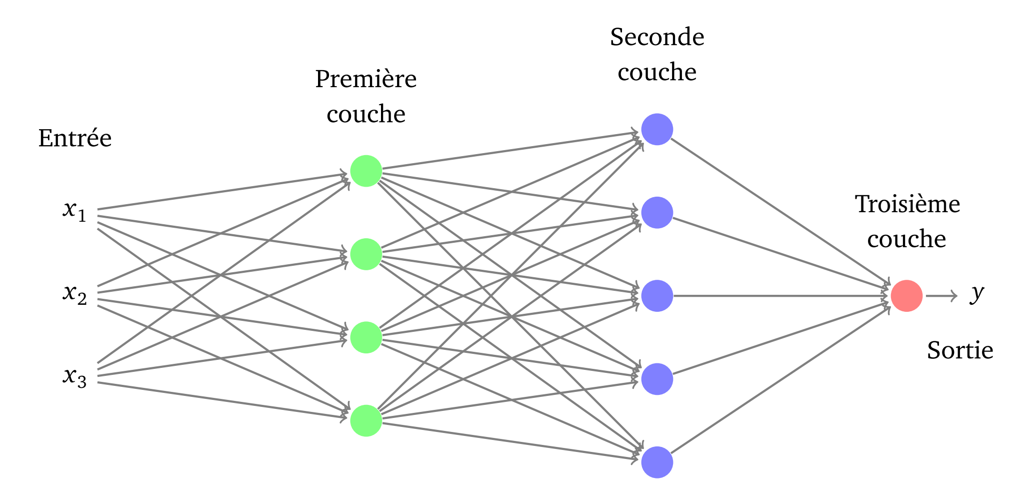


3. Réseau de neurones

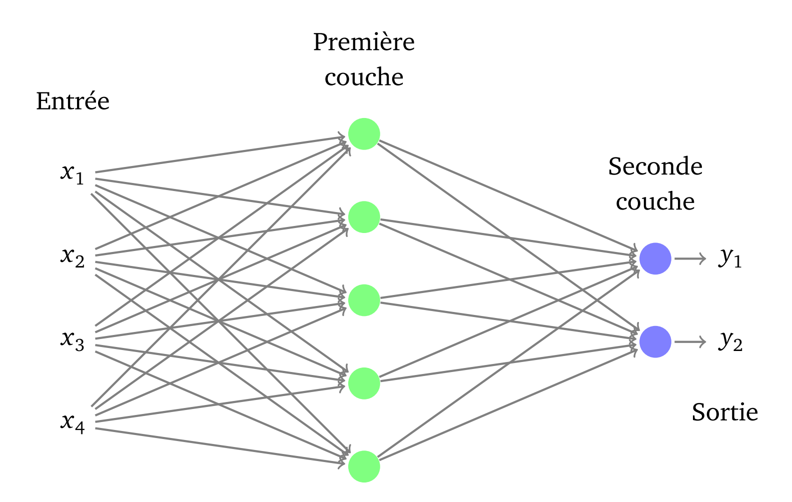
Un neurone permet de séparer l'espace en deux parties, la frontière étant linéaire (avec deux entrées c'est une droite, avec trois entrées c'est un plan. . .). Un neurone est trop élémentaire pour résoudre nos problèmes, par exemple il ne peut réaliser le « ou exclusif ». Comment faire mieux ? En connectant plusieurs neurones !

3.1. Couches de neurones

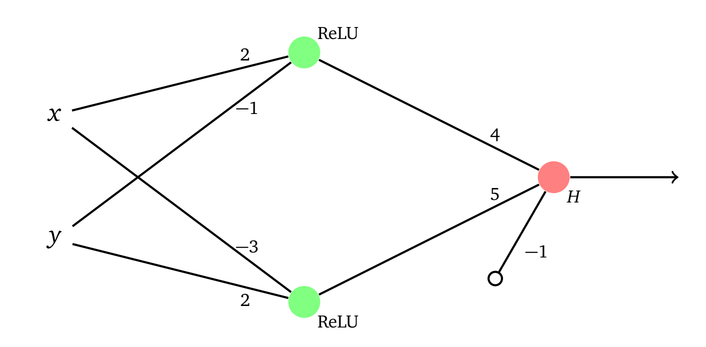
Un ***réseau de neurones*** est la juxtaposition de plusieurs neurones, regroupés par ***couches***.



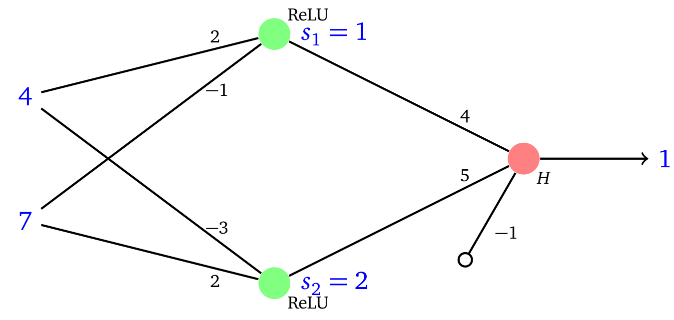
À un réseau de neurones on associe une fonction. Si à la dernière couche la fonction ne contient qu'un seul neurone (voir ci-dessus), cette fonction est $F : \mathbb{R}^n \to \mathbb{R}$, $y = F(x_1, \dots, x_n)$ où $(x_1, \dots, x_n)$ est l'***entrée*** et y est la ***sortie***. Sinon (voir ci-dessous), la fonction est $F : \mathbb{R}^n \to \mathbb{R}^p$, $(y_1, \dots, y_p) = F(x_1, \dots, x_n)$ où $(x_1, \dots, x_n) \in \mathbb{R}^n$ est l'entrée et $(y_1, \dots, y_p)$ est la sortie.



Exemple.
Voici un réseau de neurones à deux couches : 2 neurones (perceptrons linéaires) sur la première couche (ayant pour fonction d'activation la fonction ReLU), 1 neurone (perceptron affine) sur la seconde couche (de fonction d'activation H).



- Si $x = 4$ et $y = 7$ alors on calcule la sortie de chaque neurone de la première couche. Ces sorties sont les entrées du neurone de la seconde couche. Pour le premier neurone, on effectue le calcul $2 \cdot 4 + (-1) \cdot 7 = 1 \geqslant 0$, le réel étant positif la fonction ReLU le garde inchangé : le premier neurone renvoie la valeur $s_1 = 1$. Le second neurone effectue le calcul $(-3) \cdot 4 + 2 \cdot 7 = 2 \geqslant 0$ et renvoie $s_2 = 2$. Le neurone de la couche de sortie reçoit en entrées s_1 et s_2 et effectue le calcul $4 \cdot 1 + 5 \cdot 2 - 1 = 13 \geqslant 0$. La fonction d'activation étant H, ce neurone renvoie 1. Ainsi $F(4,7) = 1$.



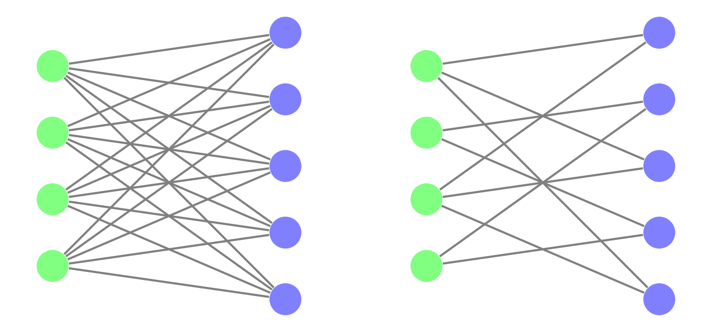
- Si $(x, y) = (3,2)$ alors $s_1 = 4$, par contre $s_2 = 0$ car $(-3) \cdot 3 + 2 \cdot 2 = -5 < 0$ et la fonction ReLU renvoie 0 (on dit que le neurone ne s'active pas). Les entrées du dernier neurone sont donc $s_1 = 4$ et $s_2 = 0$, ce neurone calcule $4 \cdot 4 + 5 \cdot 0 - 1 = 15 \geqslant 0$ et, après la fonction d'activation, renvoie 1. Donc $F(3,2) = 1$.
- Vérifier que pour $(x, y) = (\frac{1}{10}, \frac{1}{10})$ le premier neurone s'active, le second ne s'active pas (il renvoie 0) et le dernier neurone renvoie 0. Ainsi $F(\frac{1}{10}, \frac{1}{10}) = 0$.

Remarque.

- Chaque neurone est défini par ses poids et une fonction d'activation. Pour une couche donnée, on choisit toujours la même fonction d'activation. En général on choisit la même fonction d'activation pour tout le réseau, sauf peut-être pour la couche de sortie.
- Un neurone prend plusieurs valeurs en entrée mais ne renvoie qu'une seule valeur en sortie ! Si ce neurone est connecté à plusieurs autres neurones, il envoie la même valeur à tous.

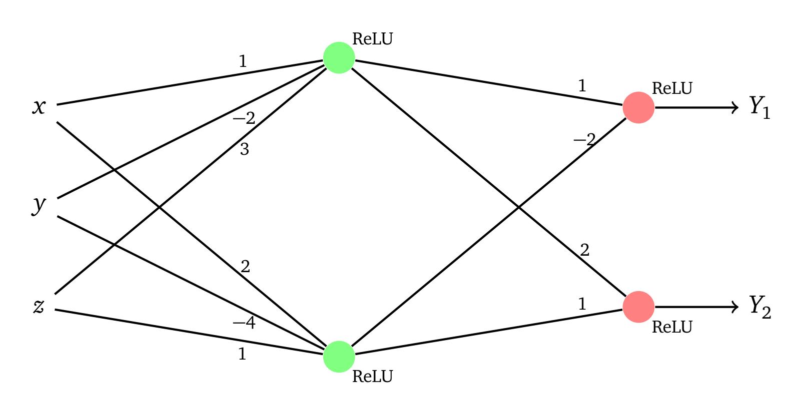


- On peut relier un neurone à tous les neurones de la couche suivante (voir ci-dessous figure de gauche). On dit que la seconde couche est ***dense*** ou ***complètement connectée***. Mais on peut aussi choisir de ne relier que certains neurones entre eux (voir ci-dessous figure de droite). S'il n'y a pas d'arêtes du neurone A vers le neurone B, c'est que la sortie de A n'intervient pas comme entrée de B (cela revient à imposer un poids nul entre ces deux neurones).



Exercice.
Pour le réseau de neurones représenté ci-dessous, calculer les valeurs de sortie (Y_1, Y_2) pour chacune des entrées (x, y, z) suivantes :

$$(0,0,0) \quad (1,0,0) \quad (1,-1,1) \quad (3,2,1)$$

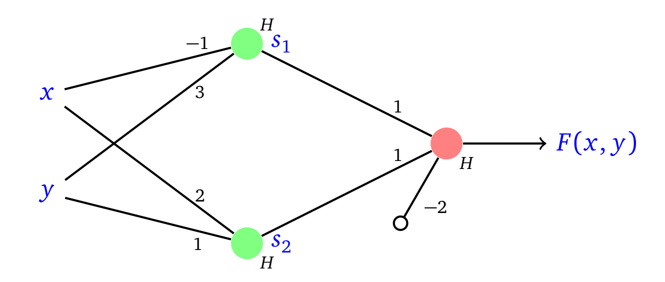


Trouver un (x, y, z) tel que $Y_1 = 1$ et $Y_2 = 7$ (commencer par déterminer les valeurs de sorties s_1 et s_2 de la première couche).
Existe-t-il (x, y, z) tel que Y_2 soit nul, mais pas Y_1 ?

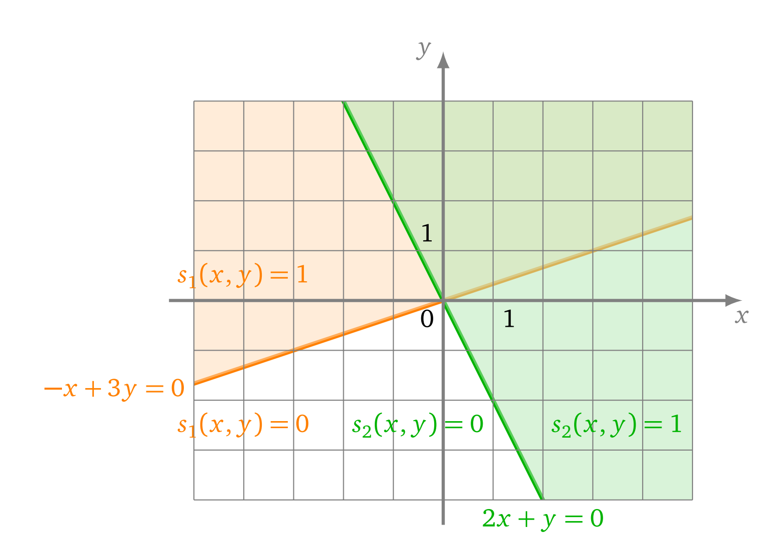
3.2. Exemples

Étudions quelques exemples plus en détails. Au lieu de calculer la sortie $F(x, y)$ pour des valeurs données, nous allons calculer $F(x, y)$ quelque soit (x, y).

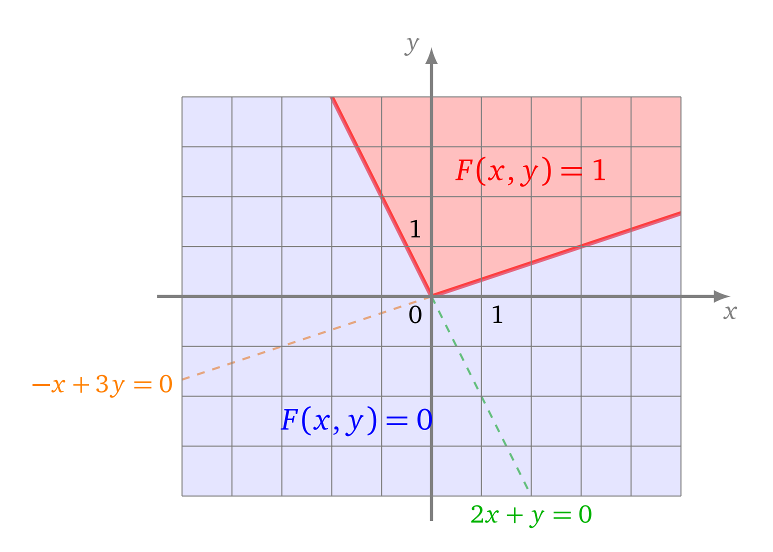
Exemple.
Voici un réseau de 3 neurones : deux sur la première couche et un sur la seconde. La fonction d'activation est partout la fonction marche de Heaviside. Combien vaut la fonction associée F selon l'entrée (x, y) ?



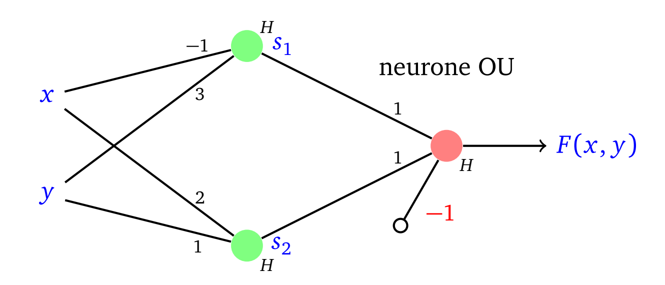
On commence par calculer les sorties des neurones de la première couche. Pour le premier neurone la sortie s_1 dépend du signe de $-x + 3y$. Si $-x + 3y \geqslant 0$ alors $s_1(x, y) = 1$, sinon $s_1(x, y) = 0$. Donc pour les points (x, y) situés au-dessus de la droite d'équation $-x + 3y = 0$, on a $s_1(x, y) = 1$. De même pour le second neurone, on a $s_2(x, y) = 1$ pour les points situés au dessus de la droite $2x + y = 0$. Voir la figure ci-dessous.



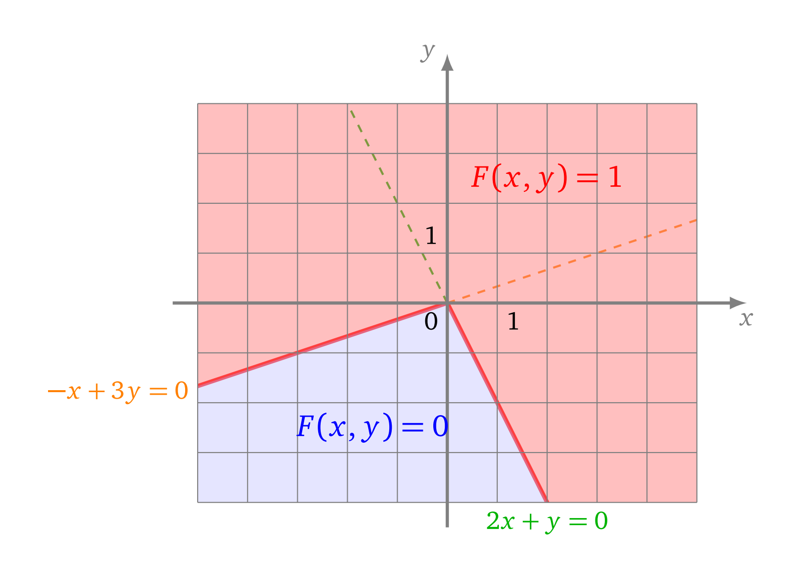
On reconnaît dans le neurone de sortie un neurone qui réalise le « ET ». C'est pourquoi l'ensemble des points pour lesquels F vaut 1 est l'intersection des deux demi-plans en lesquels s_1 et s_2 valent 1. Ainsi $F(x,y) = 1$ dans un secteur angulaire et $F(x,y) = 0$ ailleurs. Voir la figure ci-dessous.



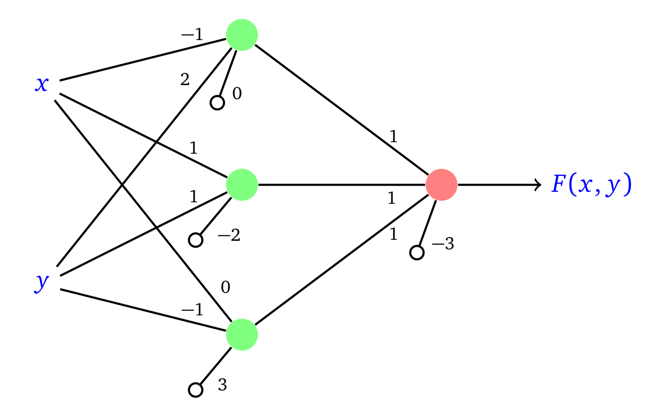
Exemple.
On reprend la même architecture que l'exemple précédent, mais en changeant les poids du neurone de sortie qui réalise cette fois l'opération « OU ».



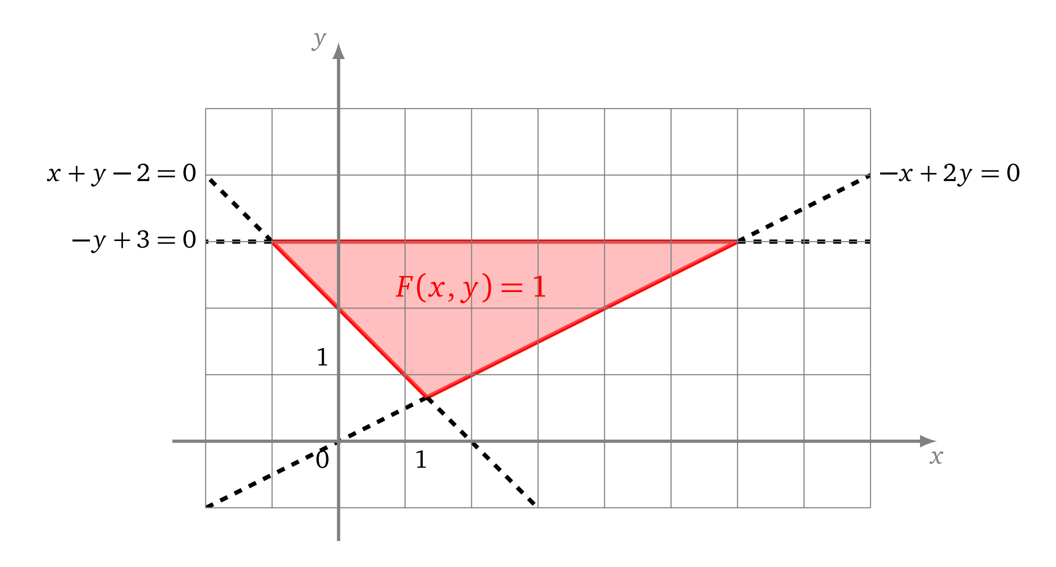
L'ensemble des points pour lesquels F vaut 1 est maintenant l'union des deux demi-plans en lesquels s_1 et s_2 valent 1.



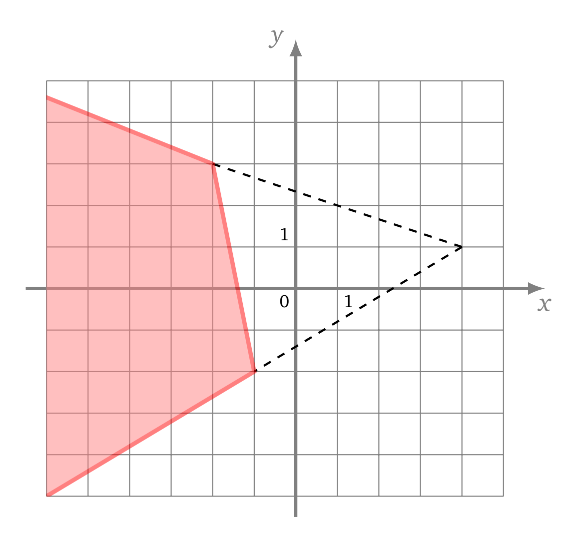
Exemple.
Voici un réseau de neurones un peu plus compliqué (la fonction d'activation est H partout).



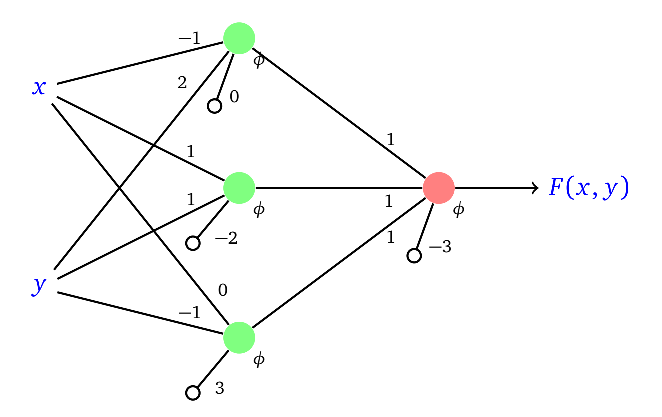
Chaque neurone de la première couche délimite un demi-plan. Ce sont les demi-plans $-x+2y \geqslant 0$, $x+y-2 \geqslant 0$ et $-y+3 \geqslant 0$.
Le neurone de sortie est un neurone qui réalise l'opération « ET » : il s'active uniquement si les trois précédents neurones sont activés. Ainsi la sortie finale $F(x,y)$ vaut 1 si et seulement si (x,y) appartient simultanément aux trois demi-plans.



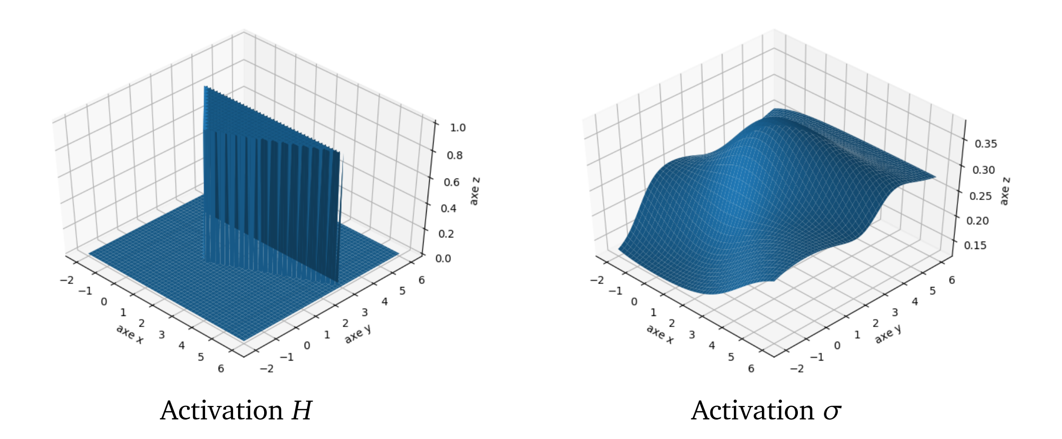
Exercice.
En utilisant les idées de l'exemple précédent et pour le dessin ci-dessous, trouver un réseau de neurones dont la fonction F vaut 1 pour la zone colorée et 0 ailleurs.



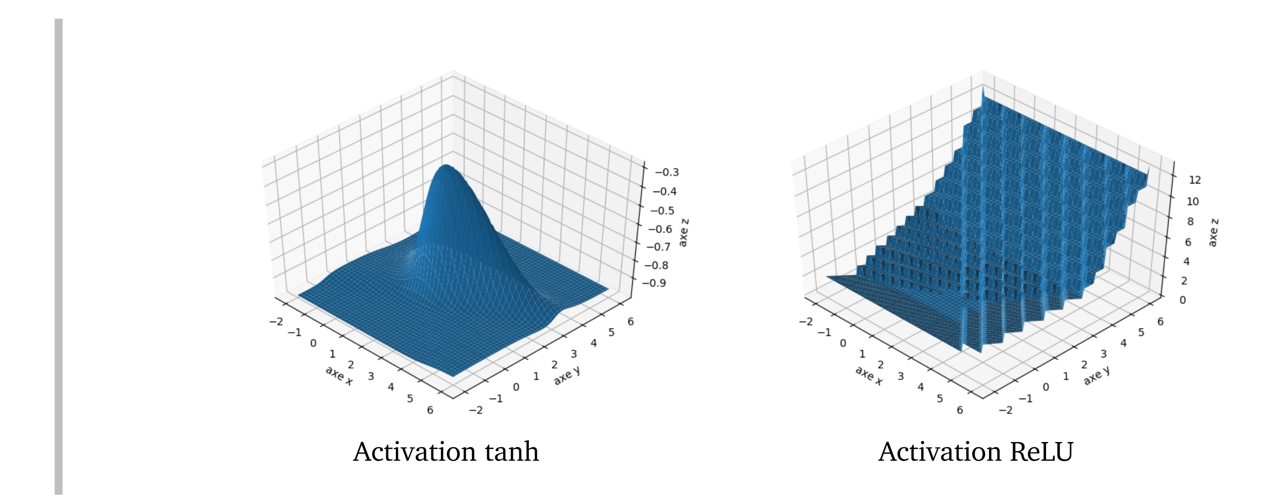
Exemple.
On termine en comparant les fonctions produites par des réseaux ayant la même architecture, les mêmes poids mais de fonction d'activation ϕ différente : H, σ, tanh et ReLU.



Voici les graphiques 3D obtenus pour les fonctions d'activation H (comme dans l'exemple vu auparavant), σ, tanh et ReLU. Les fonctions F obtenues dépendent fortement du choix de la fonction d'activation.



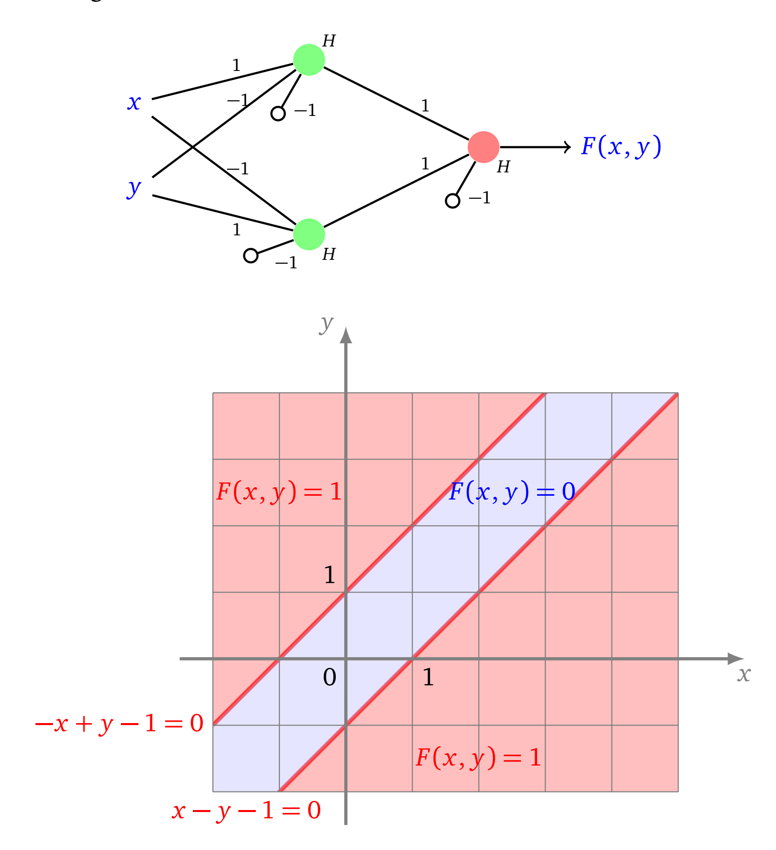
Activation H Activation σ



4. Théorie des réseaux de neurones

4.1. OU exclusif

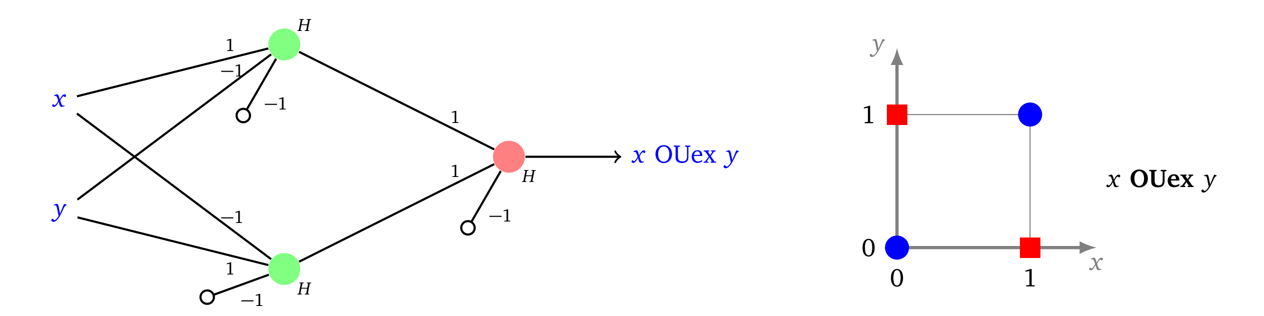
Nous avons vu qu'un seul neurone ne permet pas de réaliser la fonction associée au « OU exclusif ». Avec plusieurs neurones c'est facile ! Voici un réseau de neurones qui sépare le plan en trois parties, le secteur central en lequel la fonction associée au réseau vaut 0, alors que la fonction vaut 1 partout ailleurs (y compris sur la frontière rouge.



Ce réseau permet d'obtenir une valeur $F(x, y) = 1$ en $(1, 0)$ et $(0, 1)$ et une valeur $F(x, y) = 0$ en $(0, 0)$ et $(1, 1)$.

L'idée de ce réseau vient du fait que l'opération « OU exclusif » est une combinaison de « OU » et de « ET » :

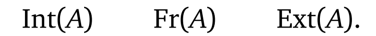
$$x \text{ OUex } y = (x \text{ ET non} y) \text{ OU } (\text{non} x \text{ ET } y).$$



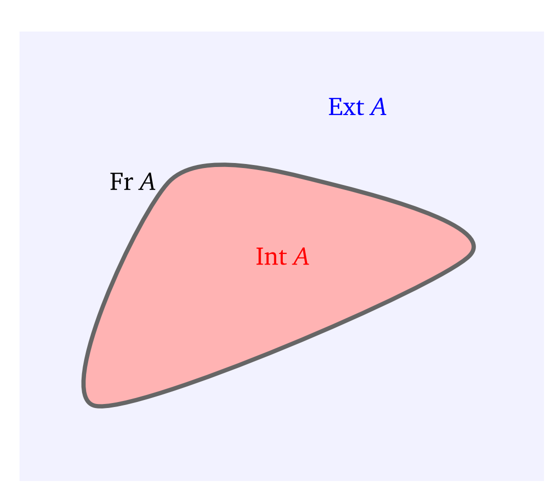
Le neurone du haut réalise « x ET non$y$ », le neurone du bas « non$x$ ET y » et celui de sortie l'opération « OU ».

4.2. Ensemble réalisable

Nous aimerions savoir quels ensembles peuvent être décrits par un réseau de neurones. On rappelle qu'un ensemble $A \subset \mathbb{R}^2$ découpe le plan en trois parties disjointes : intérieur, frontière, extérieur :



$$\text{Int}(A) \qquad \text{Fr}(A) \qquad \text{Ext}(A).$$

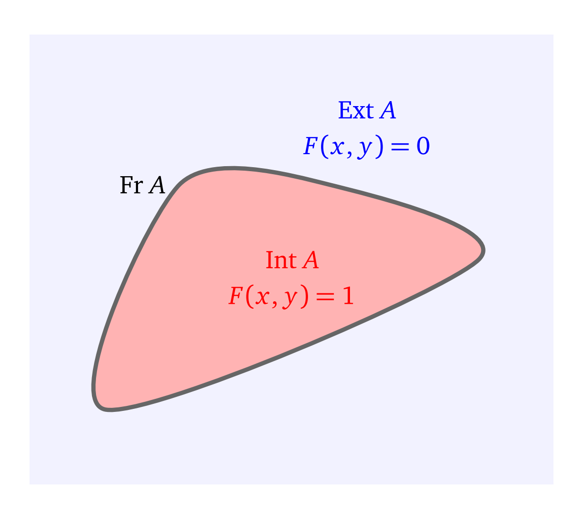


Définition.
Un ensemble A est dit ***réalisable par un réseau de neurones*** s'il existe un réseau de neurones $\mathcal{R}$ (d'unique fonction d'activation la fonction marche de Heaviside) tel que la fonction $F : \mathbb{R}^2 \to \mathbb{R}$ associée à $\mathcal{R}$ vérifie :

$$F(x, y) = 1 \quad \text{pour tout } (x, y) \in \text{Int}(A)$$

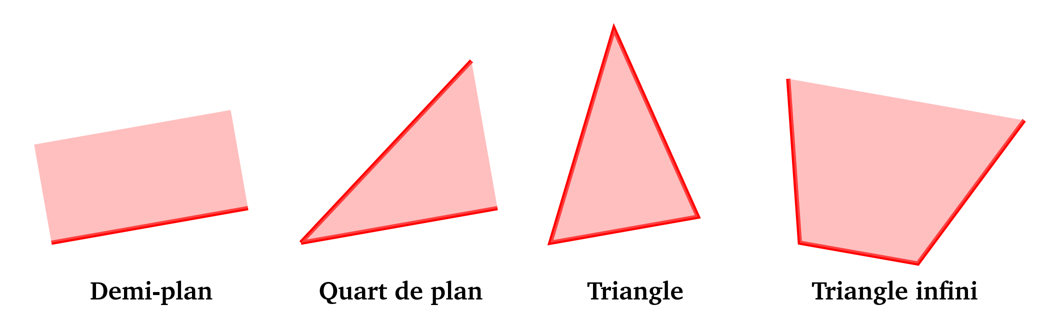
et

$$F(x, y) = 0 \quad \text{pour tout } (x, y) \in \text{Ext}(A).$$



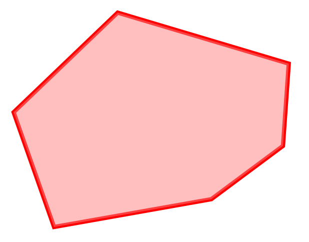
Remarque : on n'exige rien sur la frontière $\mathrm{Fr}(A)$, F peut y prendre la valeur 0 ou la valeur 1.
Voici les types d'ensembles que nous avons déjà réalisés :

- les demi-plans,
- les « quart de plans »,
- les zones triangulaires,
- les triangles avec un sommet « à l'infini ».



En augmentant le nombre de neurones, on peut réaliser les quadrilatères convexes et plus généralement n'importe quelle zone polygonale convexe.

Proposition 3.
Tout zone polygonale convexe à n côtés est réalisable par un réseau de $n+1$ neurones.



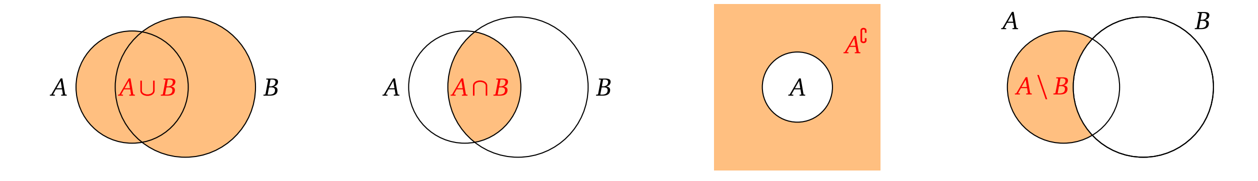
Démonstration. Un polygone convexe à n côtés est l'intersection de n demi-plans. Chaque demi-plan, bordé par une droite d'équation du type $ax+by+c=0$, correspond à un neurone de la première couche dont les coefficients sont (a,b) et le biais est c (ou alors $(-a,-b)$ et $-c$). Sur la seconde couche, on place un neurone qui réalise l'opération « ET » sur les n entrées : tous ses coefficients sont 1 et le biais est $-n$. □

Continuons avec des opérations élémentaires sur les ensembles réalisables.

Proposition 4.
Si A et B sont deux ensembles du plan réalisables par des réseaux de neurones alors :

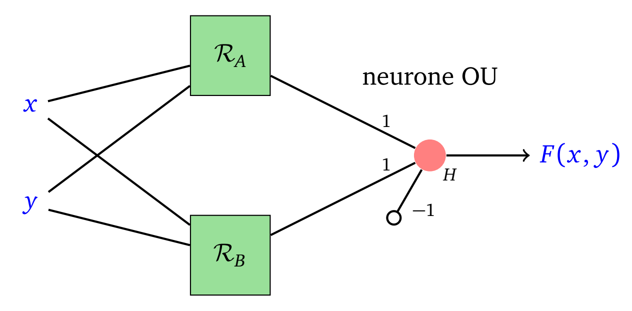
$$A \cup B \qquad A \cap B \qquad A^{\complement} \qquad A \setminus B$$

sont aussi des ensembles réalisables.

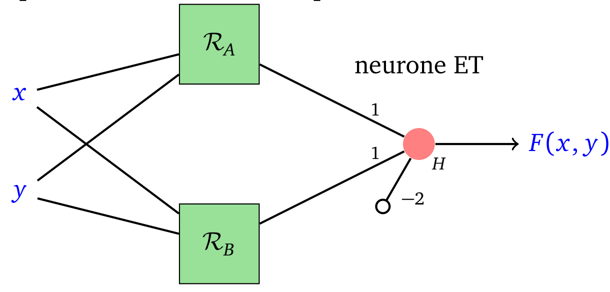


Démonstration.

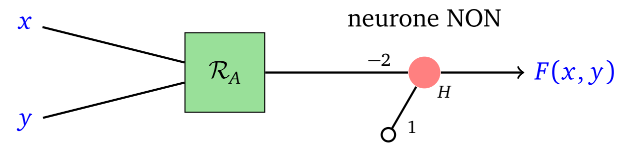
- Si A est réalisé par un réseau $\mathcal{R}_A$ et B par un réseau $\mathcal{R}_B$ alors on crée un nouveau réseau $\mathcal{R}$ en superposant $\mathcal{R}_A$ et $\mathcal{R}_B$ et en ajoutant un neurone « OU » à partir des sorties de $\mathcal{R}_A$ et $\mathcal{R}_B$. Ainsi si (x,y) est dans $A\cup B$, il est dans A ou dans B, une des sorties $\mathcal{R}_A$ ou $\mathcal{R}_B$ vaut alors 1 et le neurone sortant de $\mathcal{R}$ s'active.



- Pour réaliser $A \cap B$, on remplace le neurone « OU » par un neurone « ET ».



- Pour réaliser le complément d'un ensemble A, on utilise l'opération « non » : 0 s'envoie sur 1 et 1 sur 0. Ceci se fait par l'application $s \mapsto H(1-2s)$. Il suffit juste de rajouter un neurone « NON » à $\mathcal{R}_A$.

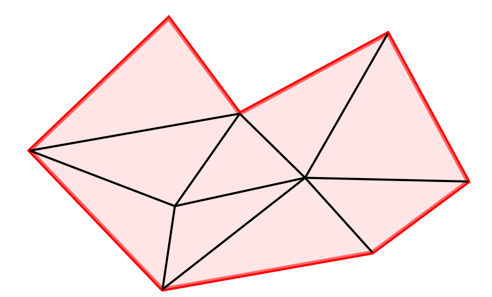


- Comme $A \setminus B = A \cap \big((A \cap B)^\complement\big)$ alors il est possible de réaliser $A \setminus B$ comme succession d'opérations élémentaires $\cap$, $\complement$ et $\cup$.

□

Proposition 5.
Tout polygone (convexe ou non) est réalisable par un réseau de neurones.

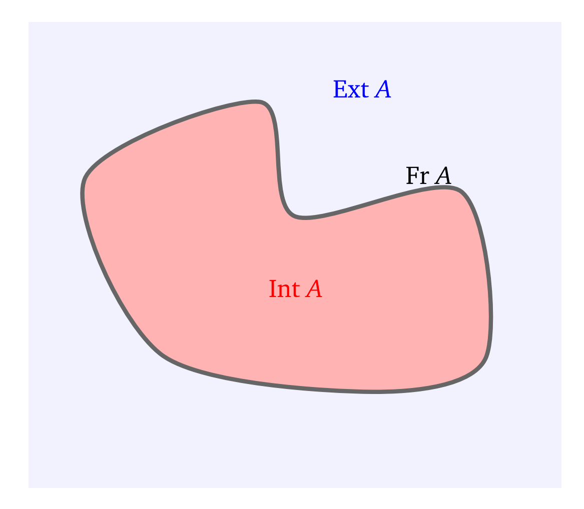
Démonstration. Un polygone peut être découpé en triangles. Chaque triangle est réalisable, donc l'union des triangles l'est aussi.



□

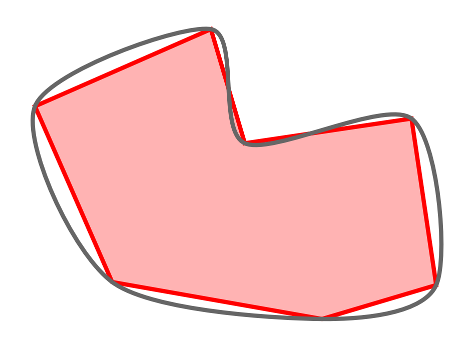
4.3. Approximation d'ensembles

On rappelle qu'une *courbe de Jordan* C est une courbe fermée simple (c'est l'image d'un cercle par une application continue injective). Le théorème suivant est une sorte de théorème d'approximation universelle géométrique.



Théorème 1.
Un ensemble A délimité par une courbe de Jordan peut être approché d'aussi près que l'on veut par un ensemble A' réalisable par un réseau de neurones.

Il s'agit juste du fait qu'une courbe de Jordan peut être approchée par un polygone. C'est un résultat théorique qui ne dit en rien comment choisir la structure du réseau ou les poids.



5. Théorème d'approximation universelle

Nous allons maintenant prouver qu'un réseau de neurones bien construit peut approcher n'importe quelle fonction.
Dans cette section nous partirons d'une seule entrée $x \in \mathbb{R}$, avec une seule sortie $y = F(x) \in \mathbb{R}$. Les réseaux de neurones de cette section produisent donc des fonctions $F : \mathbb{R} \to \mathbb{R}$.
L'objectif est le suivant : on nous donne une fonction $f : [a, b] \to \mathbb{R}$ et nous devons trouver un réseau, tel que la fonction F associée à ce réseau soit proche de f :

$$F(x) \simeq f(x) \qquad \text{pour tout } x \in [a, b].$$

Pour paramétrer le réseau nous allons bien sûr fixer des poids, mais avant cela choisissons les fonctions d'activation :

- pour le neurone de sortie, on choisit la fonction identité $\phi(x) = x$,
- pour tous les autres neurones, on choisit la fonction marche de Heaviside.

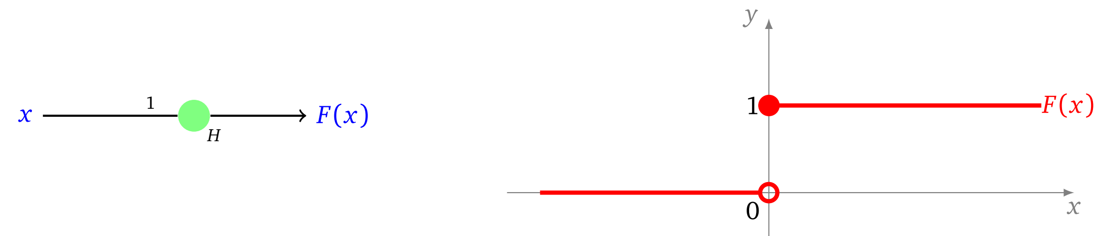
Remarque.

- Si on choisissait aussi la fonction d'activation marche de Heaviside pour le neurone de sortie, alors F ne pourrait prendre que deux valeurs, 0 ou 1, ce qui empêcherait de réaliser la plupart des fonctions.
- Par contre, si on choisissait la fonction identité pour tous les neurones alors on ne réaliserait que des fonctions affines $F(x) = ax + b$ (en effet la composition de plusieurs fonctions affines reste une fonction affine).

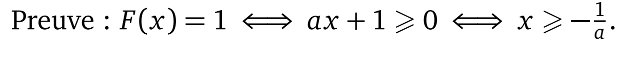
5.1. Fonctions marches

Nous allons réaliser des fonctions de plus en plus compliquées à partir d'éléments très simples.
Commençons par étudier le comportement d'un seul neurone avec la fonction d'activation marche de Heaviside (on rajoutera le neurone de sortie plus tard).
Voici différents neurones et les fonctions qu'ils réalisent :

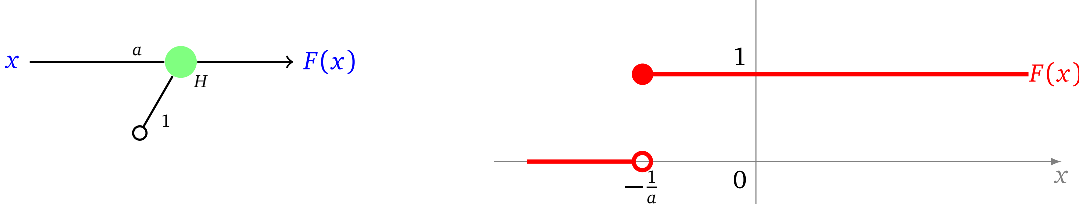
- La fonction marche de Heaviside.



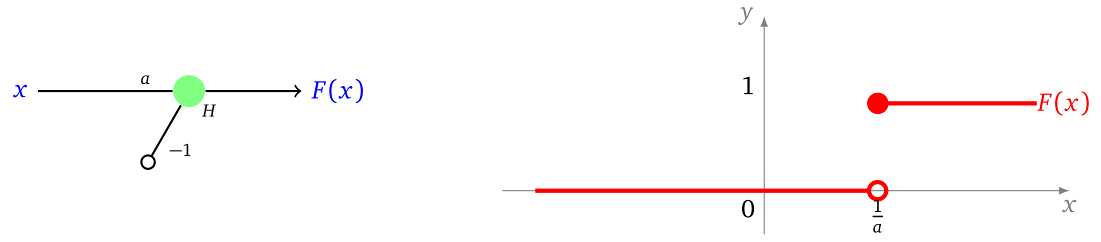
- La fonction marche décalée vers la gauche, avec $a > 0$.



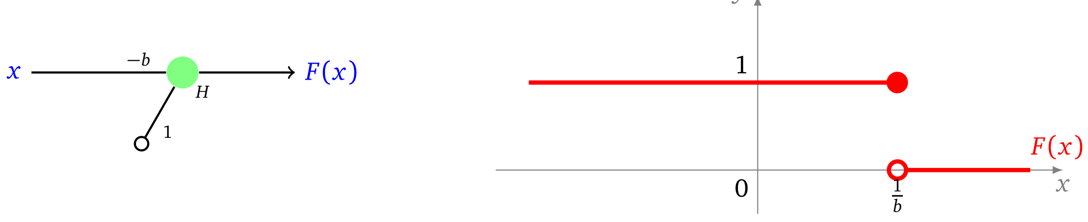
Preuve : $F(x) = 1 \iff ax + 1 \geqslant 0 \iff x \geqslant -\frac{1}{a}$.



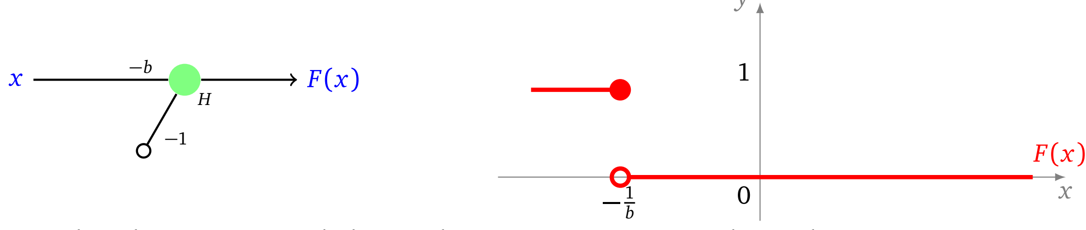
- La fonction marche décalée vers la droite, avec $a > 0$.



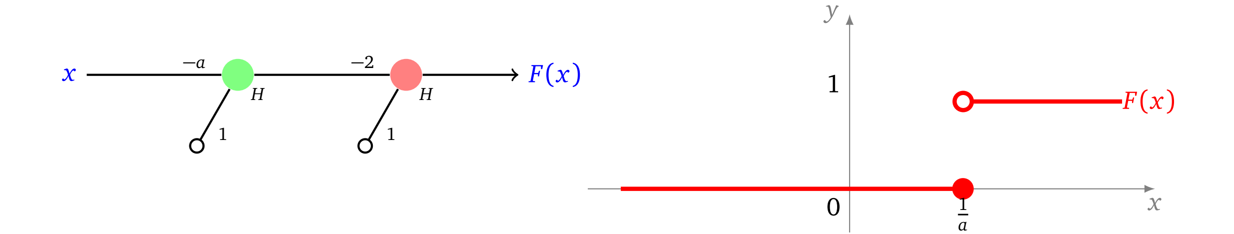
- La fonction marche à l'envers décalée vers la droite, avec $b > 0$.
  Preuve : $F(x) = 1 \iff -bx + 1 \geqslant 0 \iff bx \leqslant 1 \iff x \leqslant \frac{1}{b}$.



- La fonction marche à l'envers décalée vers la gauche, avec $b > 0$.

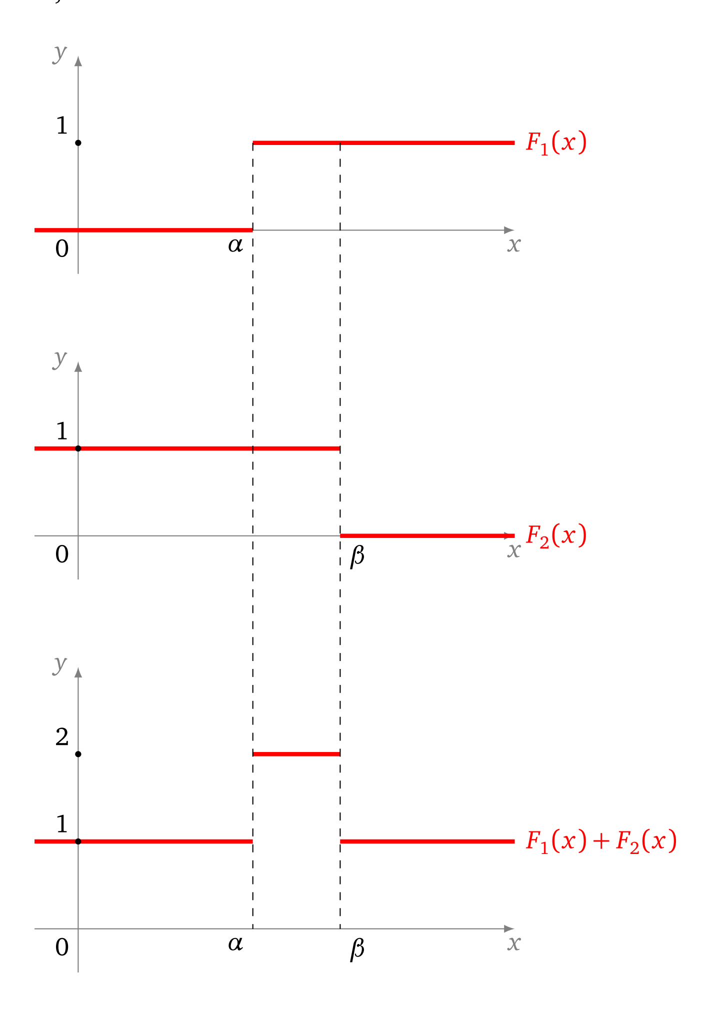


Selon les cas, la valeur au niveau de la marche est 0 ou 1. On peut obtenir l'autre situation en rajoutant un neurone de type « NON ».

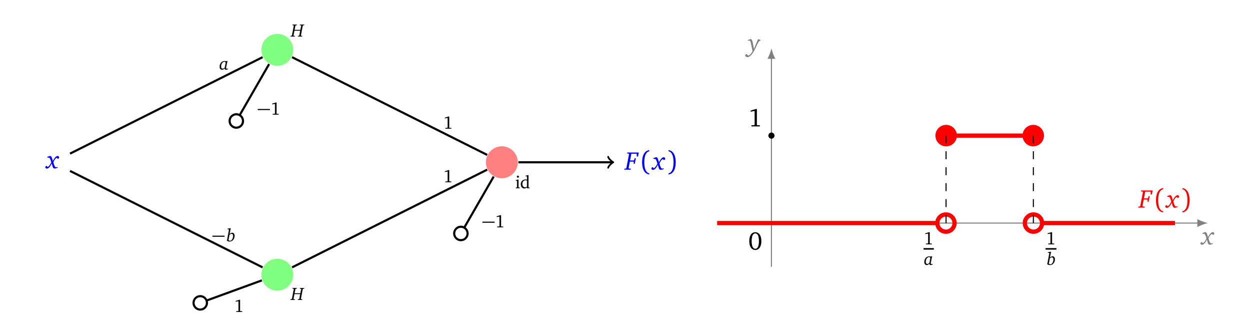


5.2. Fonctions créneaux

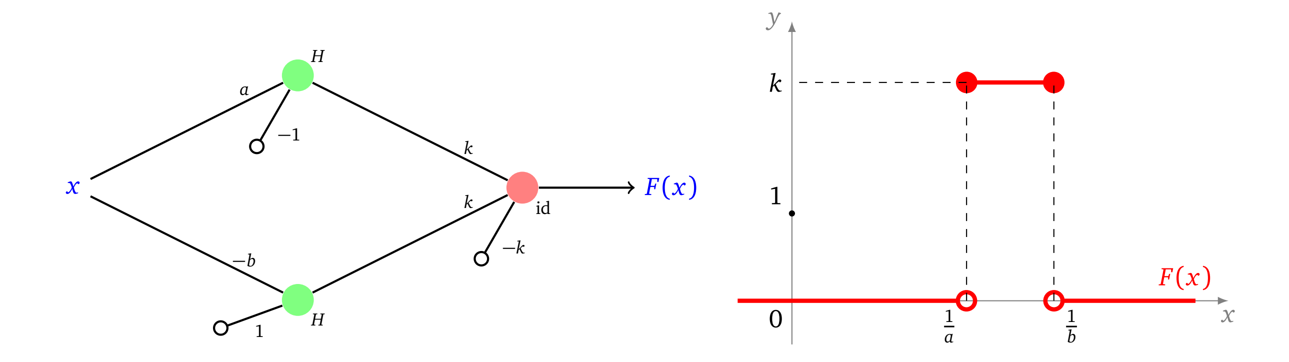
Pour réaliser un « créneau », l'idée est d'additionner une marche à l'endroit et une marche à l'envers.



Pour effectuer cette opération, nous allons construire un réseau avec deux neurones sur la première couche (de fonction d'activation H) et un neurone sur la seconde couche (de fonction d'activation identité) qui additionne les deux sorties précédentes et soustrait 1 (afin de ramener la ligne de base à 0).

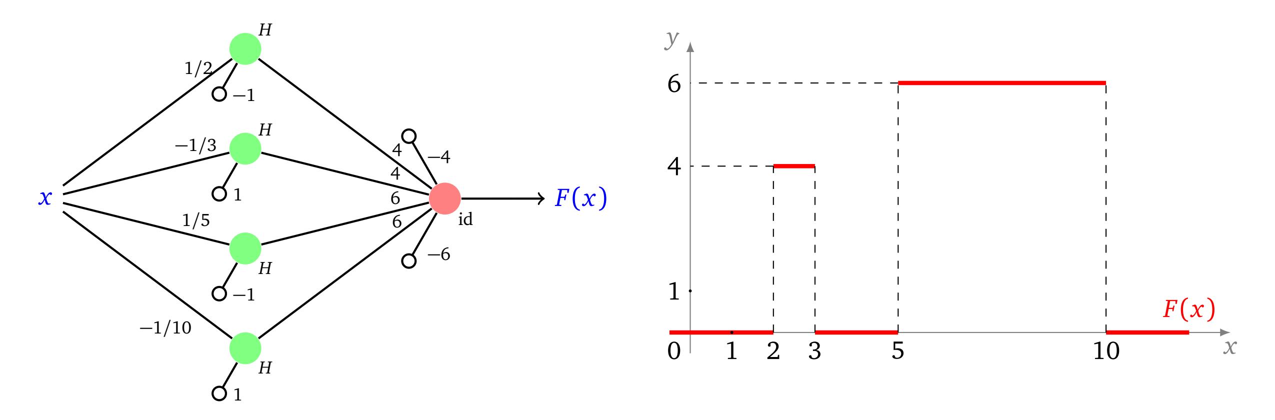


Si on veut une marche plus haute il suffit de changer les poids du neurone de sortie d'un facteur k.



5.3. Fonctions en escalier

On réalise des doubles créneaux en superposant les premières couches de chaque créneau et en réunissant les deux neurones de sortie. Voici un exemple avec un créneau de hauteur 4 sur $[2,3]$ et un créneau de hauteur 6 sur $[5,10]$.

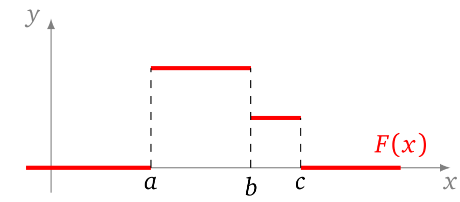


On peut aussi calculer la valeur de la fonction F de la façon suivante (s_i représente la sortie du neurone numéro i de la première couche) :

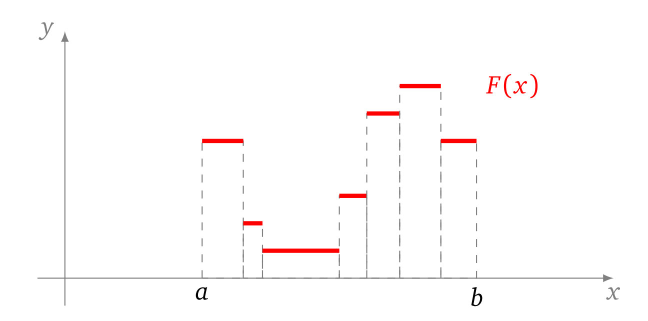
$$F(x) = \underbrace{4s_1 + 4s_2 - 4}_{\text{vaut 4 ou 0}} + \underbrace{6s_3 + 6s_4 - 6}_{\text{vaut 6 ou 0}} = \begin{cases} 4 & \text{si } x \in [2,3[\\ 6 & \text{si } x \in [5,10[\\ 0 & \text{sinon} \end{cases}$$

Remarque.

- Noter l'écriture avec deux biais -4 et -6 pour le neurone de sortie. C'est juste une écriture pour décomposer et expliquer le « vrai » biais qui est $-4-6=-10$.
- Il peut y avoir un problème pour réaliser deux créneaux contigus. Si on n'y prend pas garde, la valeur est mauvaise à la jonction (c'est la somme des deux valeurs). Pour corriger le problème, il faut utiliser une marche où on a changé la valeur au bord, voir la fin de la section 5.1.



Une *fonction en escalier* est une fonction qui est constante sur un nombre fini d'intervalles bornés.



Proposition 6.
Toute fonction en escalier est réalisable par un réseau de neurones.

Démonstration. Soit n le nombre de marches de l'escalier. On construit un réseau de $2n+1$ neurones. La première couche est constituée de n paires de neurones, chaque paire réalise une marche de l'escalier. La seconde couche contient uniquement le neurone de sortie, les coefficients sont choisis pour ajuster la hauteur de la marche et le biais assure que la fonction vaut 0 en dehors de l'escalier.
Si on veut les valeurs exactes aux bornes des marches, il faut éventuellement ajouter des neurones de type « NON » entre la première couche et le neurone de sortie. □

5.4. Théorème d'approximation universelle (une variable)

Nous pouvons maintenant énoncer le résultat théorique le plus important de ce chapitre.

Théorème 2 (Théorème d'approximation universelle)**.**
Toute fonction continue $f : [a,b] \to \mathbb{R}$ *peut être approchée d'aussi près que l'on veut par une fonction* $F : [a,b] \to \mathbb{R}$ *réalisée par un réseau de neurones.*

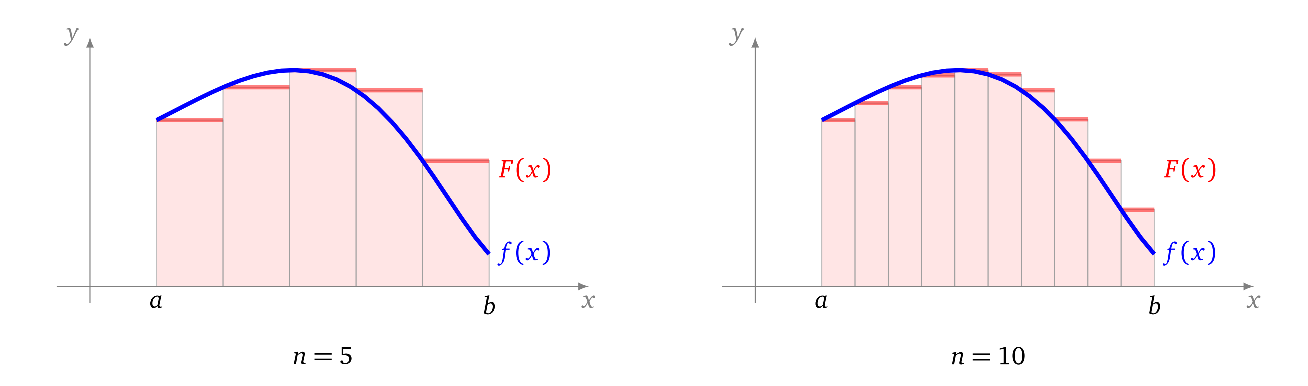
Plusieurs commentaires importants.

- Tout d'abord rappelons que nous réalisons nos neurones avec des fonctions d'activation H (marche de Heaviside) sauf pour le neurone de sortie qui a pour fonction d'activation l'identité.
- Les hypothèses sont importantes : f est continue et définie sur un intervalle fermé et borné. Si une des hypothèses venait à manquer l'énoncé serait faux.
- Bien que l'énoncé ne le dise pas, on peut concrètement réaliser à la main un réseau qui approche la fonction f (voir le chapitre suivant). Cependant, ce n'est pas l'esprit des réseaux de neurones.
- Que signifie « approcher d'aussi près que l'on veut la fonction f » ? C'est dire que pour chaque $\epsilon > 0$, il existe une fonction F (ici issue d'un réseau de neurones), telle que :

$$\text{pour tout } x \in [a,b] \qquad |f(x) - F(x)| < \epsilon.$$

  C'est l'***approximation uniforme*** des fonctions.

Démonstration. La preuve est simple : toute fonction continue sur un intervalle $[a,b]$ peut être approchée d'aussi près que l'on veut par une fonction en escalier. Par exemple, on subdivise l'intervalle $[a,b]$ en n sous-intervalles $[x_i, x_{i+1}]$ et on prend une marche de hauteur $f(x_i)$ sur cet intervalle. Comme nous avons prouvé que l'on sait réaliser toutes les fonctions en escalier, la preuve est terminée.



□

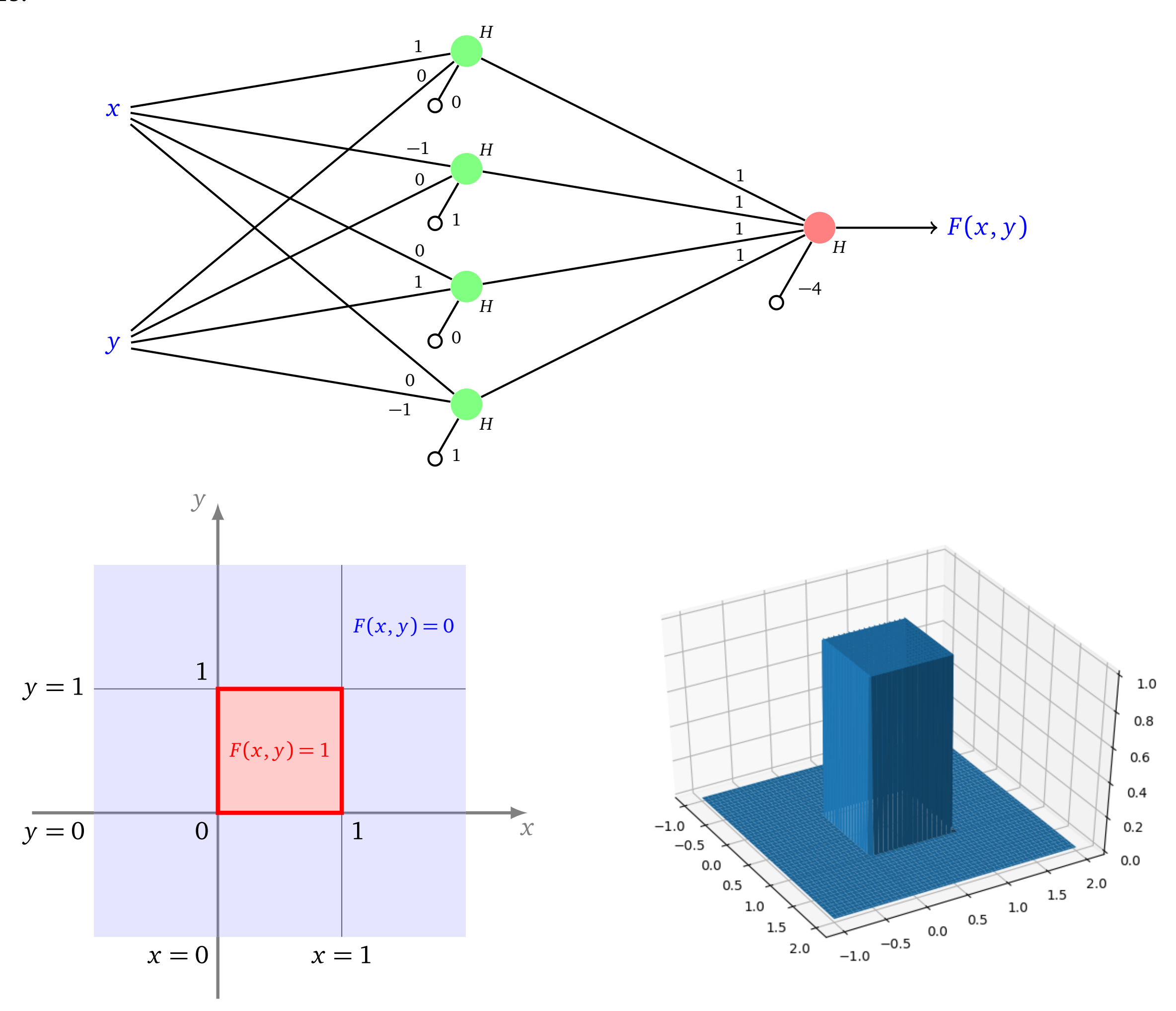
Remarque : la preuve se rapproche de la construction de l'intégrale. Pour calculer l'intégrale, on calcule en fait l'aire de rectangles. Ces rectangles correspondent à nos fonctions en escalier.

5.5. Théorème d'approximation universelle (deux variables et plus)

Ce que nous avons fait pour une variable, nous pouvons le faire pour deux variables (ou plus).

Théorème 3 (Théorème d'approximation universelle)**.**
Toute fonction continue $f : [a, b] \times [a, b] \to \mathbb{R}$ *peut être approchée d'aussi près que l'on veut par une fonction* $F : [a, b] \times [a, b] \to \mathbb{R}$ *réalisée par un réseau de neurones.*

Il suffit là encore de réaliser des fonctions marches élémentaires. Nous ne donnons pas de détails mais seulement l'exemple d'un réseau qui réalise la fonction $F : \mathbb{R}^2 \to \mathbb{R}$ qui vaut 1 sur $[0, 1] \times [0, 1]$ et 0 partout ailleurs.



Python : tensorflow avec keras - partie 1

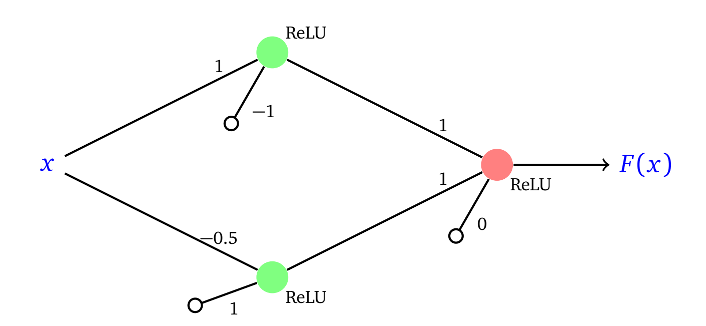
Chapitre 6

Vidéo ■ partie 6.1. Utiliser tensorflow
Vidéo ■ partie 6.2. Tensorflow : exemples à une variable
Vidéo ■ partie 6.3. Tensorflow : exemples à deux variables

Le module Python tensorflow est très puissant pour l'apprentissage automatique. Le module keras a été élaboré pour pouvoir utiliser tensorflow plus simplement. Dans cette partie nous continuons la partie facile : comment utiliser un réseau de neurones déjà paramétré ?

1. Utiliser *tensorflow* avec *keras*

Le module *keras* permet de définir facilement des réseaux de neurones en les décrivant couche par couche. Pour l'instant nous définissons les poids à la main, en attendant de voir plus tard comment les calculer avec la machine. Pour commencer nous allons créer le réseau de neurones correspondant à la figure suivante :



Ceux qui ne veulent pas s'embêter avec les détails techniques peuvent seulement lire la sous-section 1.2 car nous proposerons ensuite un outil simple dans la partie 2.

1.1. Module *keras* de *tensorflow*

En plus d'importer le module `numpy` (abrégé par `np`), il faut importer le sous-module `keras` du module `tensorflow` et quelques outils spécifiques :

```
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

1.2. Couches de neurones

Nous allons définir l'architecture d'un réseau très simple, en le décrivant couche par couche.

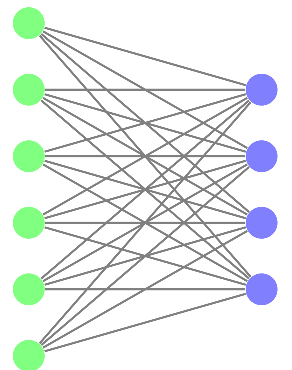
```
# Architecture du réseau
modele = Sequential()

# Couches de neurones
modele.add(Dense(2, input_dim=1, activation='relu'))
modele.add(Dense(1, activation='relu'))
```

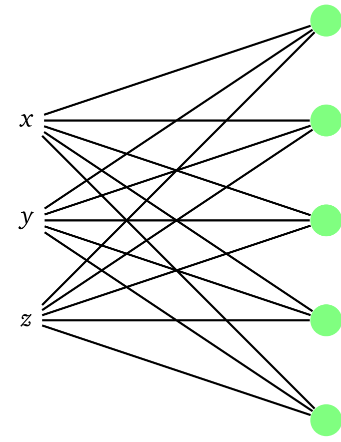
Explications :

- Notre réseau s'appelle `modele`, il est du type `Sequential`, c'est-à-dire qu'il va être décrit par une suite de couches les unes à la suite des autres.
- Chaque couche est ajoutée à la précédente par `modele.add()`. L'ordre d'ajout est donc important.
- Chaque couche est ajoutée par une commande :

  `modele.add(Dense(nb_neurones, activation=ma_fonction))`
- Une couche de type `Dense` signifie que chaque neurone de la nouvelle couche est connecté à toutes les sorties des neurones de la couche précédente.



- Pour chaque couche, il faut préciser le nombre de neurones qu'elle contient. S'il y a n neurones alors la couche renvoie n valeurs en sortie. On rappelle qu'un neurone renvoie la même valeur de sortie vers tous les neurones de la couche suivante.
- Pour la première couche, il faut préciser le nombre de valeurs en entrée (par l'option `input_dim = ...`). Dans le code ici, on a une entrée d'une seule variable. Sur la figure ci-dessous un exemple d'une entrée de dimension 3.

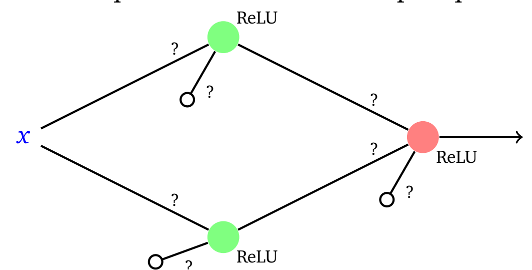


- Pour les autres couches, le nombre d'entrées est égal au nombre de sorties de la couche précédente. Il n'est donc pas nécessaire de le préciser.
- Pour chaque couche, il faut également préciser une fonction d'activation (c'est la même pour tous les neurones d'une même couche). Plusieurs fonctions d'activation sont prédéfinies :

  `'relu'` (ReLU), `'sigmoid'` (σ), `'linear'` (identité).

Nous verrons plus tard comment définir notre propre fonction d'activation, comme par exemple la fonction marche de Heaviside

- Notre exemple ne possède qu'une entrée et comme il n'y a qu'un seul neurone sur la dernière couche alors il n'y a qu'une seule valeur en sortie. Ainsi notre réseau va définir une fonction $F : \mathbb{R} \to \mathbb{R}$, $x \mapsto F(x)$.
- Mais attention, pour l'instant ce n'est qu'un *modèle* de réseau puisque nous n'avons pas fixé de poids.



- Pour vérifier que tout va bien jusque là, on peut exécuter la commande

`modele.summary()`

qui affiche un résumé des couches et du nombre de poids à définir.

1.3. Les poids

Lors de la définition d'un réseau et de la structure de ses couches, des poids aléatoires sont attribués à chaque neurone. La démarche habituelle est ensuite d'entraîner le réseau, automatiquement, afin qu'il trouve de « bons » poids. Mais pour l'instant, nous continuons de fixer les poids de chaque neurone à la main.

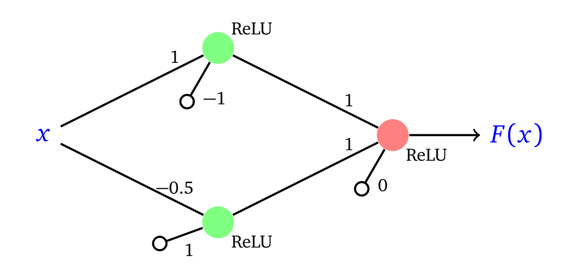
La commande pour fixer les poids est `set_weights()`.

Voici la définition des poids de la première couche, numérotée 0 :

```
# Couche 0
coeff = np.array([[1.,-0.5]])
biais = np.array([-1,1])
poids = [coeff,biais]
modele.layers[0].set_weights(poids)
```

Définissons les poids de la couche numéro 1 :

```
# Couche 1
coeff = np.array([[1.0],[1.0]])
biais = np.array([0])
poids = [coeff,biais]
modele.layers[1].set_weights(poids)
```



Voici quelques précisions concernant la commande `set_weights()`. Son utilisation n'est pas très aisée.

- Les poids sont définis pour tous les éléments d'une couche, par une commande `set_weights(poids)`.
- Les poids sont donnés sous la forme d'une liste : `poids = [coeff,biais]`.
- Les biais sont donnés sous la forme d'un vecteur de biais (un pour chaque neurone).
- Les coefficients sont donnés sous la forme d'un tableau à deux dimensions. Ils sont définis par entrée. Attention, la structure n'est pas naturelle (nous y reviendrons).

Pour vérifier que les poids d'une couche sont corrects, on utilise la commande `get_weights()`, par exemple pour la première couche :

`modele.layers[0].get_weights()`

Cette instruction renvoie les poids sous la forme d'une liste [coefficients,biais] du type :

`[ [[ 1. -0.5]], [-1. 1.] ]`

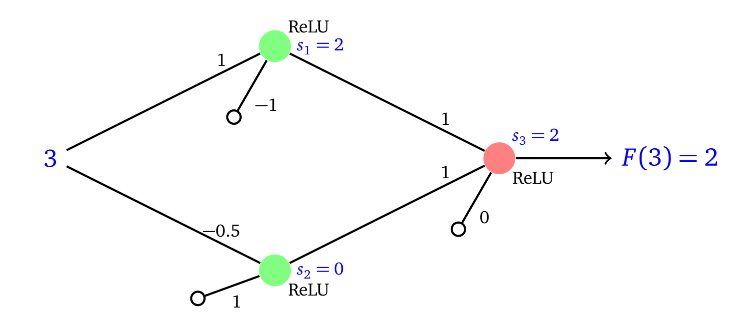
Astuce ! Cette commande est aussi très pratique avant même de fixer les poids, pour savoir quelle est la forme que doivent prendre les poids afin d'utiliser `set_weights()`.

1.4. Évaluation

Comment utiliser le réseau ? C'est très simple avec `predict()`. Notre réseau définit une fonction $x \mapsto F(x)$. L'entrée correspond donc à un réel et la sortie également. Voici comment faire :

```
entree = np.array([[3.0]])
sortie = modele.predict(entree)
```

Ici `sortie` vaut `[[2.0]]` et donc $F(3) = 2$. Ce que l'on peut vérifier à la main en calculant les sorties de chaque neurone.



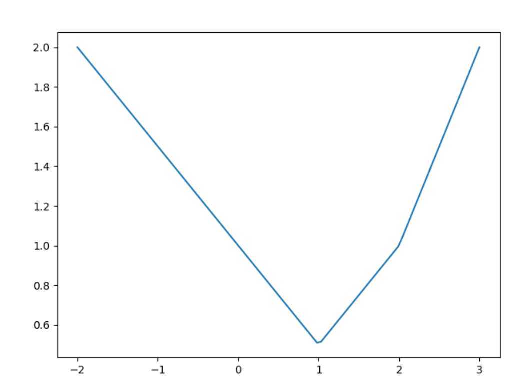
1.5. Visualisation

Afin de tracer le graphe de la fonction $F : \mathbb{R} \to \mathbb{R}$, on peut calculer d'autres valeurs :

```
import matplotlib.pyplot as plt
liste_x = np.linspace(-2, 3, num=100)
entree =  np.array([[x] for x in liste_x])

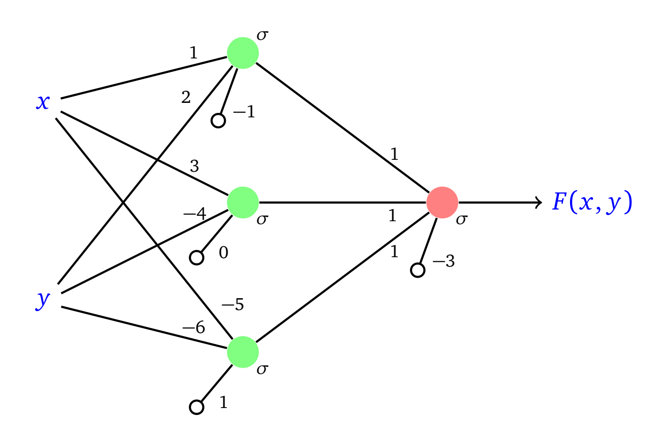
sortie = modele.predict(entree)

liste_y = np.array([y[0] for y in sortie])
plt.plot(liste_x,liste_y)
plt.show()
```



1.6. Autre exemple

Créons le réseau de neurones ci-dessous :



Couches.

```
# Architecture du réseau
modele = Sequential()

# Couches de neurones
modele.add(Dense(3, input_dim=2, activation='sigmoid'))
modele.add(Dense(1, activation='sigmoid'))
```

La première couche possède 3 neurones, chacun ayant deux entrées. La seconde couche n'a qu'un seul neurone (qui a automatiquement 3 entrées). La fonction d'activation est partout la fonction σ.

Poids.

```
# Couche 0
coeff = np.array([[1.0,3.0,-5.0],[2.0,-4.0,-6.0]])
biais = np.array([-1.0,0.0,1.0])
poids = [coeff,biais]
modele.layers[0].set_weights(poids)
```

Remarquez que les poids ne sont pas définis neurone par neurone, mais par entrée : d'abord les poids de la première entrée pour chaque neurone, puis les poids de la seconde entrée pour chaque neurone, etc.

```
# Couche 1
coeff = np.array([[1.0],[1.0],[1.0]])
biais = np.array([-3.0])
poids = [coeff,biais]
modele.layers[1].set_weights(poids)
```

Évaluation.

```
entree = np.array([[7,-5]])
sortie = modele.predict(entree)
```

Cette fois l'entrée est du type (x, y) et la sortie correspond à un seul réel $F(x, y)$. Ici $F(7,-5) \simeq 0.123$.

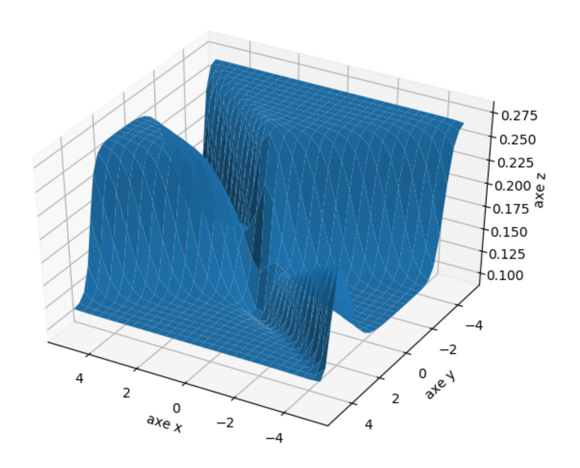
Graphique. Voici comment tracer le graphe de $F : \mathbb{R}^2 \to \mathbb{R}$. C'est un peu trop technique, ce n'est pas la peine d'en comprendre les détails.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

VX = np.linspace(-5, 5, 20)
VY = np.linspace(-5, 5, 20)
X,Y = np.meshgrid(VX, VY)
entree = np.c_[X.ravel(), Y.ravel()]

sortie = modele.predict(entree)
Z = sortie.reshape(X.shape)

fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z)
plt.show()
```



2. Exemples à une variable

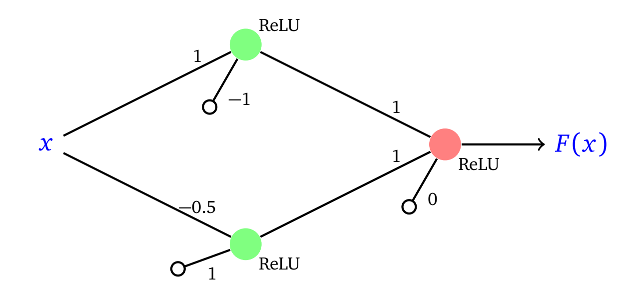
Comme la philosophie de l'apprentissage automatique n'est pas de définir les poids à la main et que ce cours ne cherche pas à trop entrer dans les détails techniques, nous avons créé spécialement pour vous un autre module :

`keras_facile`

afin de définir facilement les poids d'un réseau, d'évaluer des entrées et de tracer le graphe de la fonction associée. Ce module est téléchargeable sur le site du livre avec tous les autres codes sources.

2.1. Exemple simple

Reprenons l'exemple du réseau de la première partie.




```
from keras_facile import *

modele = Sequential()
modele.add(Dense(2, input_dim=1, activation='relu'))
modele.add(Dense(1, activation='relu'))

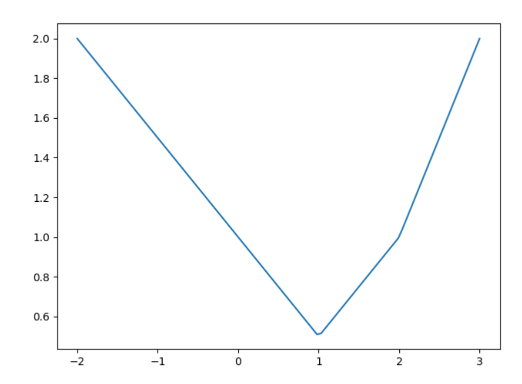
# Poids de la couche 0
# definir_poids(modele,couche,rang,coeff,biais)
definir_poids(modele,0,0,1,-1)
definir_poids(modele,0,1,-0.5,1)
affiche_poids(modele,0)          # affiche poids de la couche 0

# Poids de la couche 1
definir_poids(modele,1,0,[1,1],0)
affiche_poids(modele,1)

# Evaluation
entree = 3
sortie = evaluation(modele,entree)
print('Entrée :',entree,'Sortie :',sortie)

# Affichage graphique
affichage_evaluation_une_var(modele,-2,3)
```

C'est tout de même plus simple qu'auparavant ! Bien sûr les résultats sont les mêmes. Ce programme affiche la valeur $F(3) = 2$ et trace directement le graphe de F.



Le code commence comme précédemment en définissant l'architecture du réseau et les couches. Ce qui change et qui est plus simple :

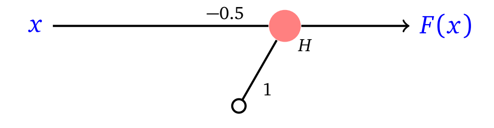
- La fonction
  `definir_poids(modele,couche,rang,coeff,biais)`
  permet de définir les poids d'un neurone à la main. Le neurone est identifié par sa couche, son rang dans cette couche, les coefficients (un nombre a ou une liste de nombres `[a1,a2]`, `[a1,a2,a3]`, etc.) et enfin le biais (un nombre).
- On vérifie que les poids sont corrects avec `affiche_poids(modele,couche)`. On peut aussi fixer tous les poids à zéro avec la fonction `poids_a_zeros(modele,couche)`.
- Une fonction d'évaluation
  `evaluation(modele,entree)`
  qui prend en entrées le modèle et un nombre (ou une liste de nombres) et renvoie un nombre. C'est une variante simpliste de `modele.predict(entree)`.
- Une seule commande permet de tracer le graphe !

2.2. La fonction marche de Heaviside

Le module `keras_facile` définit la fonction marche de Heaviside, que l'on peut ensuite utiliser lors de la déclaration de la couche par l'option :

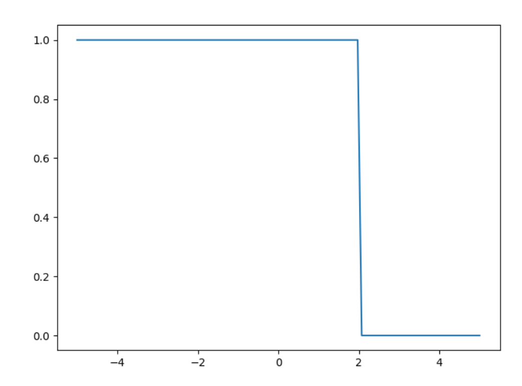
`activation = heaviside`

Attention, il n'y a pas de guillemets autour de `heaviside`.
Voici un réseau d'un seul neurone avec la fonction marche de Heaviside :




```
modele = Sequential()
modele.add(Dense(1, input_dim=1, activation=heaviside))
definir_poids(modele,0,0,-0.5,1)
```

Voici le graphe de la fonction produite par ce neurone :



2.3. Théorème d'approximation universelle

Enfin, le module `keras_facile` propose une fonction `calcul_approximation()` qui rend effectif le théorème d'approximation universelle pour les fonctions d'une variable (voir le chapitre « Réseau de neurones »).
Prenons l'exemple de la fonction $f : [2, 10] \to \mathbb{R}$ définie par

$$f(x) = \cos(2x) + x\sin(3x) + \sqrt{x}$$

que l'on souhaite approcher par un réseau de neurones.
On définit d'abord la fonction f, l'intervalle $[a, b]$ et la précision souhaitée n (l'intervalle $[a, b]$ sera divisé en n sous-intervalles). Prenons par exemple $a = 2$, $b = 10$, $n = 20$.

```
def f(x):
    return np.cos(2*x) + x*np.sin(3*x) + x**0.5
```

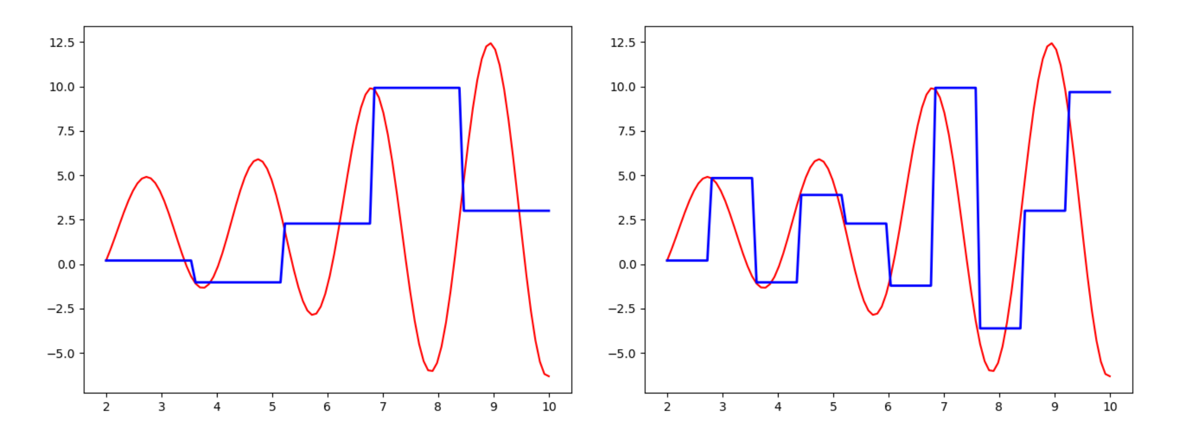
Ensuite, on définit un réseau de neurones ayant deux couches, la première avec $2n$ neurones, la seconde avec un seul neurone. Encore une fois, on renvoie au chapitre « Réseau de neurones » pour les explications. Ensuite la fonction `calcul_approximation()` calcule les poids qui conviennent pour approcher f.

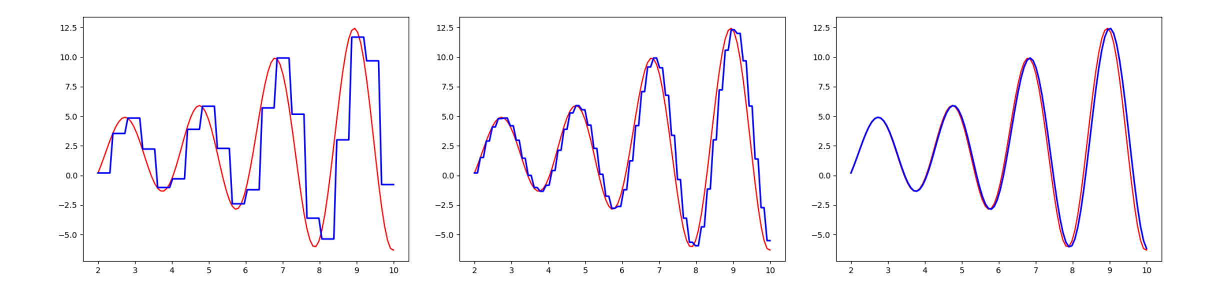
```
modele = Sequential()

modele.add(Dense(2*n,input_dim=1,activation=heaviside))
modele.add(Dense(1,activation='linear'))

calcul_approximation(modele,f,a,b,n) # calcule et définit les poids
affichage_approximation(modele,f,a,b)
```

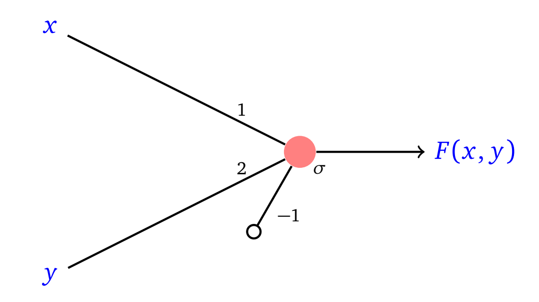
Une autre fonction trace le graphe de f (en rouge) et la fonction F issue du réseau. Ci-dessous avec $n = 5, 10, 20, 50, 100$.





3. Exemples à deux variables

3.1. Un exemple avec un seul neurone



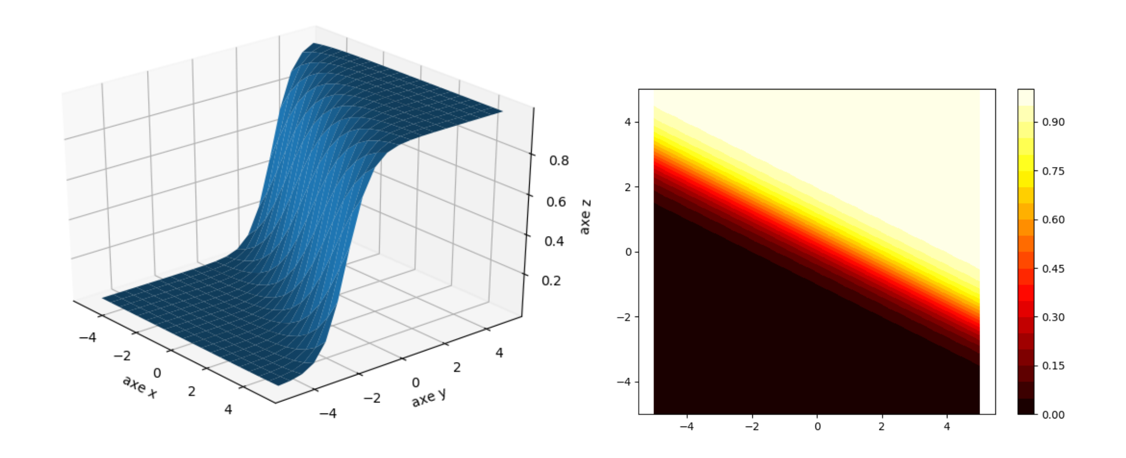
Voici la définition de ce neurone :

```
modele = Sequential()
modele.add(Dense(1, input_dim=2, activation='sigmoid'))
definir_poids(modele,0,0,[1,2],-1)
entree = [2.0,1.0]
sortie = evaluation(modele,entree)
```

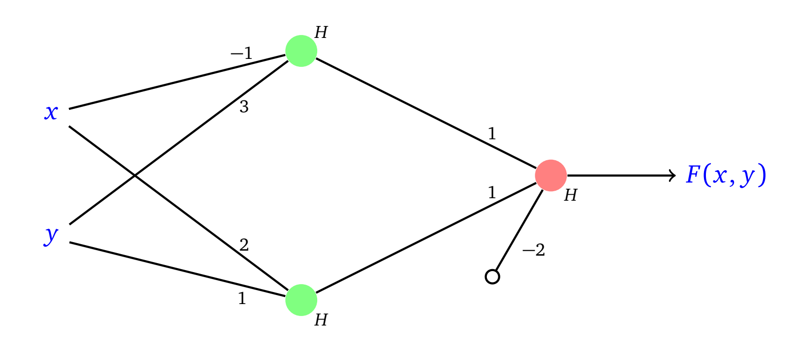
Ici l'entrée est de dimension 2 du type (x, y). Cela se définit avec `input_dim=2`, il y a donc deux coefficients à définir pour le neurone (ici `[1,2]`) ainsi qu'un biais (ici -1). Enfin, on évalue la fonction F sur un exemple, ici cela donne $F(2,1) \simeq 0.952$.

On termine en affichant le graphe de F dans l'espace et ses lignes de niveau dans le plan, pour (x, y) dans $[-5,5] \times [-5,5]$.

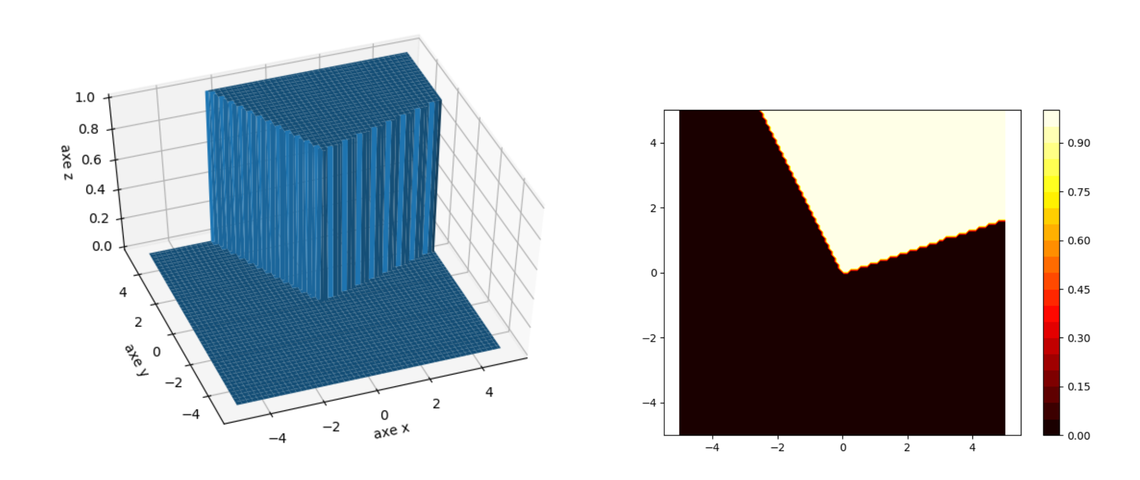
```
affichage_evaluation_deux_var_3d(modele,-5,5,-5,5)
affichage_evaluation_deux_var_2d(modele,-5,5,-5,5)
```



3.2. Un exemple avec deux couches

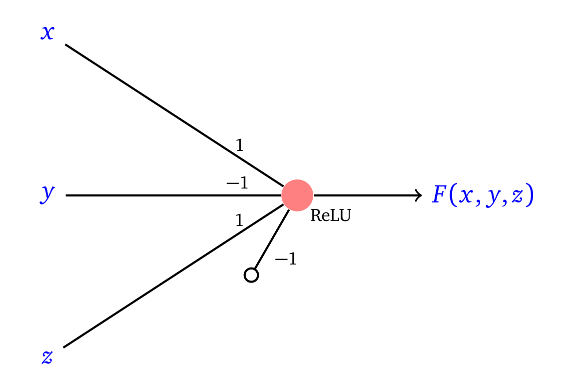



```
modele = Sequential()
modele.add(Dense(2, input_dim=2, activation=heaviside))
modele.add(Dense(1, activation=heaviside))
# Couche 0
definir_poids(modele,0,0,[-1,3],0)
definir_poids(modele,0,1,[2,1],0)
# Couche 1
definir_poids(modele,1,0,[1,1],-2)
```



4. Exemples à trois variables

4.1. Un seul neurone



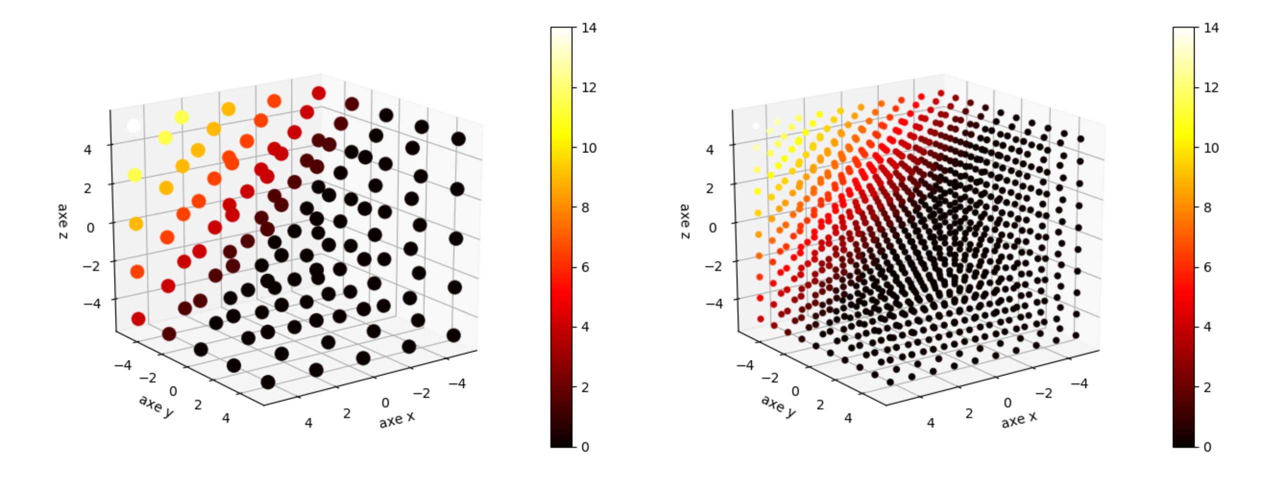
```
modele = Sequential()
modele.add(Dense(1, input_dim=3, activation='relu'))

definir_poids(modele,0,0,[1,-1,1],-1)
affiche_poids(modele,0)

entree = [2.0,3.0,4.0]
sortie = evaluation(modele,entree)

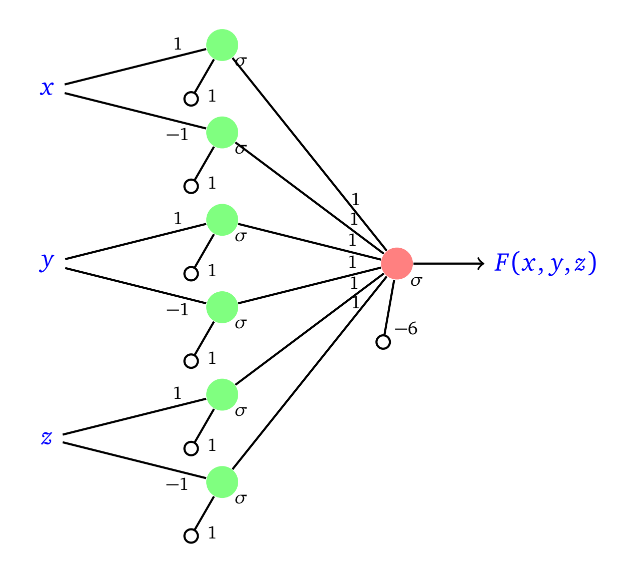
affichage_evaluation_trois_var(modele,-5,5,-5,5,-5,5,num=10)
```

Ici on obtient $F(2,3,4) = 2$.
L'affichage 3D est plus compliqué à interpréter. Chaque point représente une entrée (x, y, z), la valeur $F(x, y, z)$ est représentée par la couleur du point. La barre des couleurs donne une idée des valeurs. Ci-dessous les exemples pour $n = 5$ et $n = 10$ sous-intervalles.

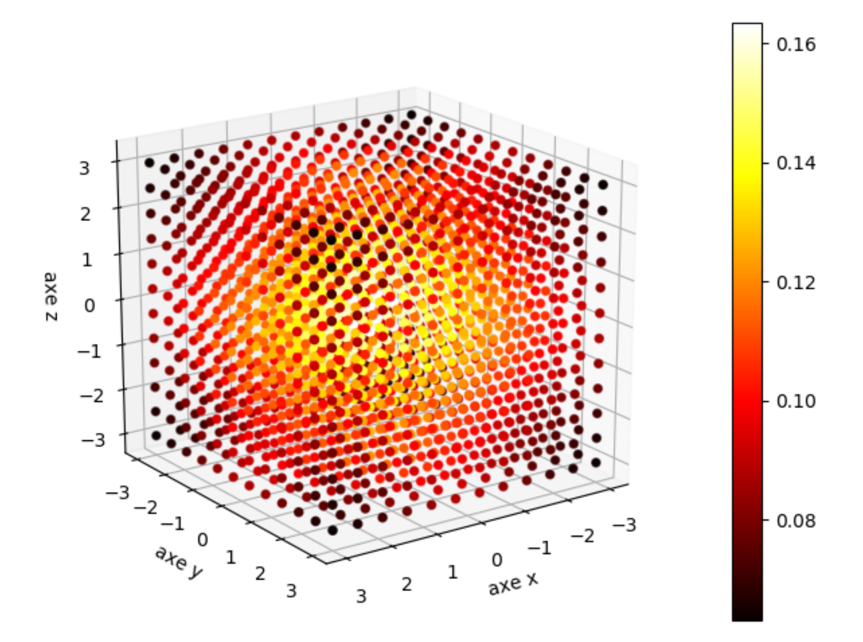


4.2. Un réseau pour réaliser un cube

Voici un réseau plus compliqué et sa représentation graphique (l'absence d'arête signifie un poids nul) :



La fonction $F(x, y, z)$ prend des valeurs élevées juste autour du point $(0,0,0)$ et des valeurs basses ailleurs (comme une sorte de Soleil positionné en $(0,0,0)$). On note que si on avait choisi la fonction de Heaviside comme fonction d'activation alors la fonction F aurait valu 1 sur le cube $[-1,1] \times [-1,1] \times [-1,1]$ et 0 ailleurs. La fonction d'activation σ « lisse » les marches.


axe z
axe y
axe x
3
2
1
0
−1
−2
−3
0.16
0.14
0.12
0.10
0.08

Chapitre 7

Gradient

Vidéo ■ partie 7.1. Dérivées partielles
Vidéo ■ partie 7.2. Gradient et géométrie
Vidéo ■ partie 7.3. Gradient et minimum/maximum
Vidéo ■ partie 7.4. Différentiation automatique
Vidéo ■ partie 7.5. Gradient pour un réseau de neurones

Le gradient est un vecteur qui remplace la notion de dérivée pour les fonctions de plusieurs variables. On sait que la dérivée permet de décider si une fonction est croissante ou décroissante. De même, le vecteur gradient indique la direction dans laquelle la fonction croît ou décroît le plus vite. Nous allons voir comment calculer de façon algorithmique le gradient grâce à la « différentiation automatique ».

1. Dérivées partielles

Pour une fonction de plusieurs variables, il existe une dérivée pour chacune des variables, qu'on appelle dérivée partielle.

1.1. Définition

Définition.
Soit $f : \mathbb{R}^2 \to \mathbb{R}$. La ***dérivée partielle*** $\frac{\partial f}{\partial x}(x_0, y_0)$ de f par rapport à la variable x au point $(x_0, y_0) \in \mathbb{R}^2$ est la dérivée en x_0 de la fonction d'une variable $x \mapsto f(x, y_0)$.
De même $\frac{\partial f}{\partial y}(x_0, y_0)$ est la dérivée partielle de f par rapport à la variable y au point (x_0, y_0).
Comme d'habitude et sauf mention contraire, nous supposerons que toutes les dérivées partielles existent.
Autrement dit, en revenant à la définition de la dérivée comme une limite :

$$\frac{\partial f}{\partial x}(x_0, y_0) = \lim_{h \to 0} \frac{f(x_0 + h, y_0) - f(x_0, y_0)}{h} \quad \text{et} \quad \frac{\partial f}{\partial y}(x_0, y_0) = \lim_{h \to 0} \frac{f(x_0, y_0 + h) - f(x_0, y_0)}{h}.$$

Plus généralement, pour une fonction $f : \mathbb{R}^n \to \mathbb{R}$ de plusieurs variables, $\frac{\partial f}{\partial x_i}(x_1, \ldots, x_n)$ est la dérivée partielle de f par rapport à la variable x_i au point $(x_1, \ldots, x_n) \in \mathbb{R}^n$. C'est la dérivée en x_i de la fonction d'une variable $x_i \mapsto f(x_1, \ldots, x_n)$ où l'on considère fixes les variables x_j pour $j \neq i$.

Notations.

$$\frac{\partial f}{\partial x}(x, y) \quad \text{et} \quad \frac{\partial f}{\partial y}(x, y)$$

sont les analogues de l'écriture $\frac{\mathrm{d}f}{\mathrm{d}x}(x)$ pour l'écriture de la dérivée lorsqu'il n'y a qu'une seule variable. Le symbole « ∂ » se lit « d rond ». Une autre notation est $\partial_x f(x, y)$, $\partial_y f(x, y)$ ou bien encore $f'_x(x, y)$, $f'_y(x, y)$.

Remarque.
Pour une fonction d'une variable $f : \mathbb{R} \to \mathbb{R}$, on distingue le nombre dérivé $f'(x_0)$ et la fonction dérivée f' définie par $x \mapsto f'(x)$. Il en est de même avec les dérivées partielles. Pour $f : \mathbb{R}^2 \to \mathbb{R}$:

- $\frac{\partial f}{\partial x}(x_0, y_0)$ et $\frac{\partial f}{\partial y}(x_0, y_0)$ sont des nombres réels.
- $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial y}$ sont des fonctions de deux variables, par exemple :

$$\frac{\partial f}{\partial x} : \begin{array}{ccc} \mathbb{R}^2 & \longrightarrow & \mathbb{R} \\ (x,y) & \longmapsto & \frac{\partial f}{\partial x}(x,y) \end{array}$$

1.2. Calculs

La calcul d'une dérivée partielle n'est pas plus compliqué que le calcul d'une dérivée.
Méthode. Pour calculer une dérivée partielle par rapport à une variable, il suffit de dériver par rapport à cette variable en considérant les autres variables comme des constantes.

Exemple.
Calculer les dérivées partielles de la fonction $f : \mathbb{R}^2 \to \mathbb{R}$ définie par $f(x,y) = x^2 e^{3y}$.

Solution.
Pour calculer la dérivée partielle $\frac{\partial f}{\partial x}$, par rapport à x, on considère que y est une constante et on dérive $x^2 e^{3y}$ comme si c'était une fonction de la variable x uniquement :

$$\frac{\partial f}{\partial x}(x,y) = 2xe^{3y}.$$

Pour l'autre dérivée $\frac{\partial f}{\partial y}$, on considère que x est une constante et on dérive $x^2 e^{3y}$ comme si c'était une fonction de y :

$$\frac{\partial f}{\partial y}(x,y) = 3x^2 e^{3y}.$$

Exemple.
Pour $f : \mathbb{R}^3 \to \mathbb{R}$ définie par $f(x,y,z) = \cos(x+y^2)e^{-z}$ on a :

$$\frac{\partial f}{\partial x}(x,y,z) = -\sin(x+y^2)e^{-z},$$
$$\frac{\partial f}{\partial y}(x,y,z) = -2y\sin(x+y^2)e^{-z},$$
$$\frac{\partial f}{\partial z}(x,y,z) = -\cos(x+y^2)e^{-z}.$$

Exemple.
Soit $f : \mathbb{R}^n \to \mathbb{R}$ définie par $f(x_1,\ldots,x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$, alors pour $i = 1,\ldots,n$:

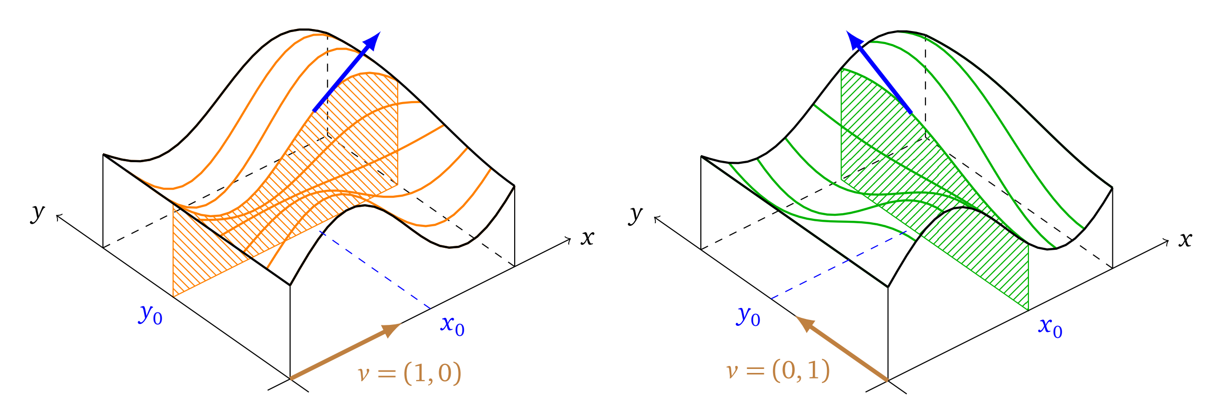
$$\frac{\partial f}{\partial x_i}(x_1,\ldots,x_n) = 2x_i.$$

1.3. Interprétation géométrique

Pour une fonction d'une variable, la dérivée est la pente de la tangente au graphe de la fonction (le graphe étant alors une courbe). Pour une fonction de deux variables $(x,y) \mapsto f(x,y)$, les dérivées partielles indiquent les pentes au graphe de f selon certaines directions (le graphe étant ici une surface). Plus précisément :

- $\frac{\partial f}{\partial x}(x_0,y_0)$ est la pente du graphe de f en (x_0,y_0) suivant la direction de l'axe (Ox). En effet cette pente est celle de la tangente à la courbe $z = f(x,y_0)$ et est donnée par la dérivée de $x \mapsto f(x,y_0)$ en x_0, c'est donc bien $\frac{\partial f}{\partial x}(x_0,y_0)$.
- $\frac{\partial f}{\partial y}(x_0,y_0)$ est la pente du graphe de f en (x_0,y_0) suivant la direction de l'axe (Oy).

Sur la figure de gauche, la dérivée partielle $\frac{\partial f}{\partial x}$ indique la pente de la tranche parallèle à l'axe (Ox) (en orange). Sur la figure de droite, la dérivée partielle $\frac{\partial f}{\partial y}$ indique la pente de la tranche parallèle à l'axe (Oy) (en vert).



2. Gradient

Le gradient est un vecteur dont les coordonnées sont les dérivées partielles. Il a de nombreuses applications géométriques car il donne l'équation des tangences aux courbes et surfaces de niveau. Surtout, il indique la direction dans laquelle la fonction varie le plus vite.

2.1. Définition

Définition.
Soit $f : \mathbb{R}^2 \to \mathbb{R}$ une fonction admettant des dérivées partielles. Le ***gradient*** de f en $(x_0,y_0) \in \mathbb{R}^2$, noté $\operatorname{grad} f(x_0,y_0)$, est le vecteur :

$$\operatorname{grad} f(x_0,y_0) = \begin{pmatrix} \dfrac{\partial f}{\partial x}(x_0,y_0) \\ \dfrac{\partial f}{\partial y}(x_0,y_0) \end{pmatrix}.$$

Les physiciens et les anglo-saxons notent souvent $\nabla f(x,y)$ pour $\operatorname{grad} f(x,y)$. Le symbole ∇ se lit « nabla ».
Plus généralement, pour $f : \mathbb{R}^n \to \mathbb{R}$, le gradient de f en $(x_1,\ldots,x_n) \in \mathbb{R}^n$ est le vecteur :

$$\operatorname{grad} f(x_1,\ldots,x_n) = \begin{pmatrix} \dfrac{\partial f}{\partial x_1}(x_1,\ldots,x_n) \\ \vdots \\ \dfrac{\partial f}{\partial x_n}(x_1,\ldots,x_n) \end{pmatrix}.$$

Exemple.

- $f(x,y) = x^2y^3$, $\operatorname{grad} f(x,y) = \begin{pmatrix} 2xy^3 \\ 3x^2y^2 \end{pmatrix}$. Au point $(x_0,y_0) = (2,1)$, $\operatorname{grad} f(2,1) = \begin{pmatrix} 4 \\ 12 \end{pmatrix}$.
- $f(x,y,z) = x^2\sin(yz)$, $\operatorname{grad} f(x,y,z) = \begin{pmatrix} 2x\sin(yz) \\ x^2z\cos(yz) \\ x^2y\cos(yz) \end{pmatrix}$.
- $f(x_1,\ldots,x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$, $\operatorname{grad} f(x_1,\ldots,x_n) = \begin{pmatrix} 2x_1 \\ \vdots \\ 2x_n \end{pmatrix}$.

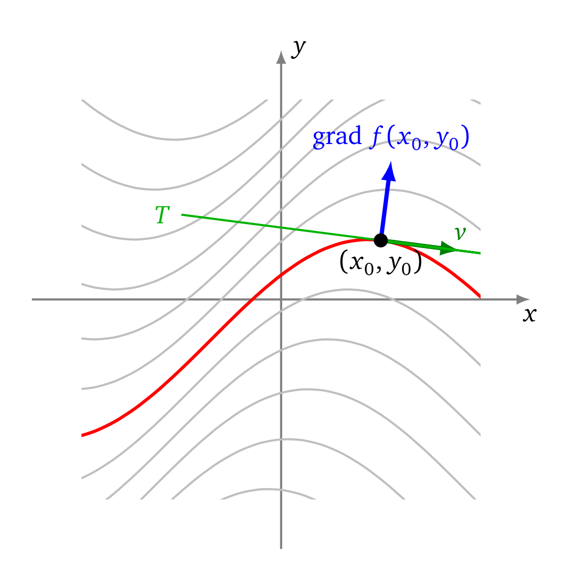
Remarque.
Le gradient est un élément de $\mathbb{R}^n$ écrit comme un vecteur colonne. Parfois, pour alléger l'écriture, on peut aussi l'écrire sous la forme d'un vecteur ligne.

2.2. Tangentes aux lignes de niveau

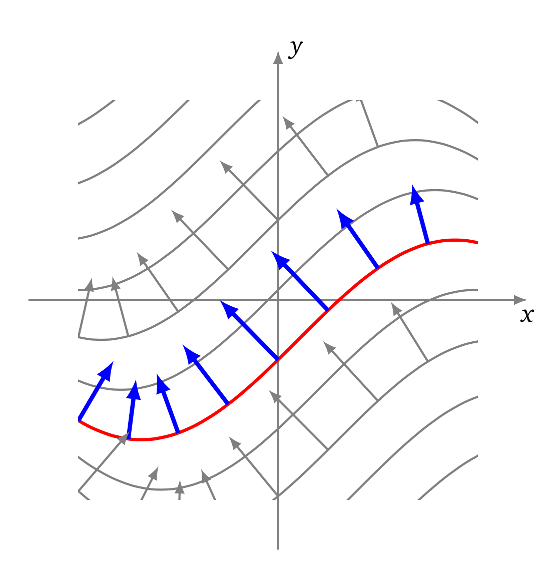
Soit $f : \mathbb{R}^2 \to \mathbb{R}$ une fonction différentiable. On considère les lignes de niveau $f(x,y) = k$.

Proposition 1.
Le vecteur gradient $\operatorname{grad} f(x_0,y_0)$ *est orthogonal à la ligne de niveau de* f *passant au point* (x_0,y_0).

Sur ce premier dessin, sont dessinés la ligne de niveau passant par le point (x_0,y_0) (en rouge), un vecteur tangent v en ce point et la tangente à la ligne de niveau (en vert). Le vecteur gradient est un vecteur du plan qui est orthogonal à la ligne de niveau en ce point (en bleu).



À chaque point du plan, on peut associer un vecteur gradient. Ce vecteur gradient est orthogonal à la ligne de niveau passant par ce point. Nous verrons juste après comment savoir s'il est orienté « vers le haut » ou « vers le bas ».



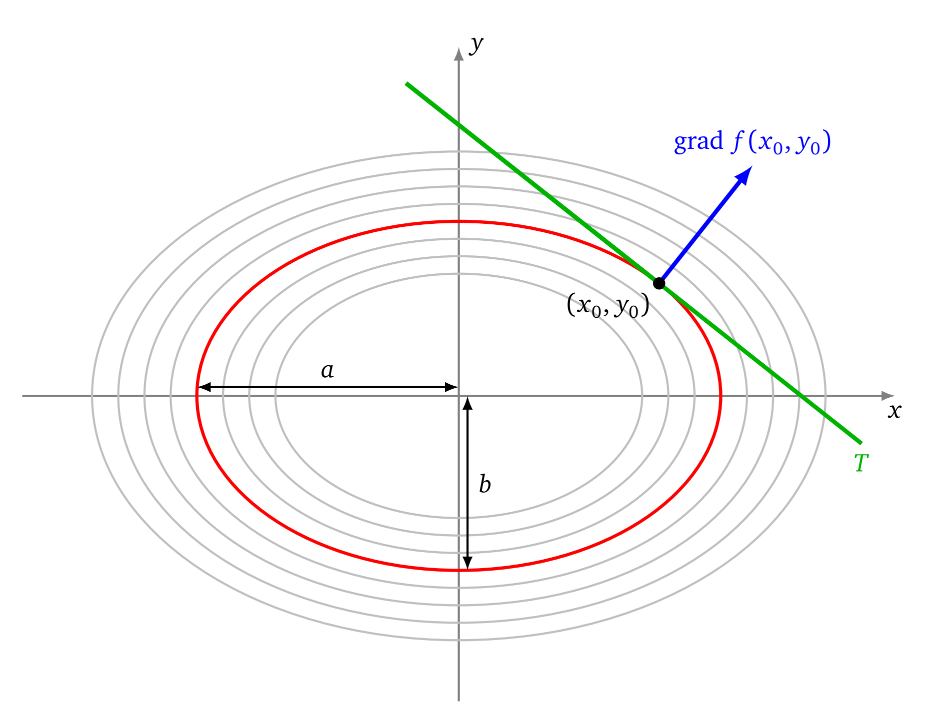
Dans le cadre de notre étude, nous nous intéressons à l'équation de la tangente.

Proposition 2.
Au point (x_0, y_0)*, l'équation de la tangente à la ligne de niveau de* f *est :*

$$\frac{\partial f}{\partial x}(x_0, y_0)(x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0)(y - y_0) = 0$$

pourvu que le gradient de f *en ce point ne soit pas le vecteur nul.*

Exemple (Tangentes à une ellipse)**.**
Trouver les tangentes à l'ellipse $\mathcal{E}$ d'équation $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$.



Cette ellipse $\mathcal{E}$ est la ligne de niveau $f(x, y) = 1$ de la fonction $f(x, y) = \frac{x^2}{a^2} + \frac{y^2}{b^2}$. Les dérivées partielles en (x_0, y_0) sont :

$$\frac{\partial f}{\partial x}(x_0, y_0) = \frac{2x_0}{a^2} \qquad \text{et} \qquad \frac{\partial f}{\partial y}(x_0, y_0) = \frac{2y_0}{b^2}.$$

L'équation de la tangente à l'ellipse $\mathcal{E}$ en ce point est donc :

$$\frac{2x_0}{a^2}(x - x_0) + \frac{2y_0}{b^2}(y - y_0) = 0.$$

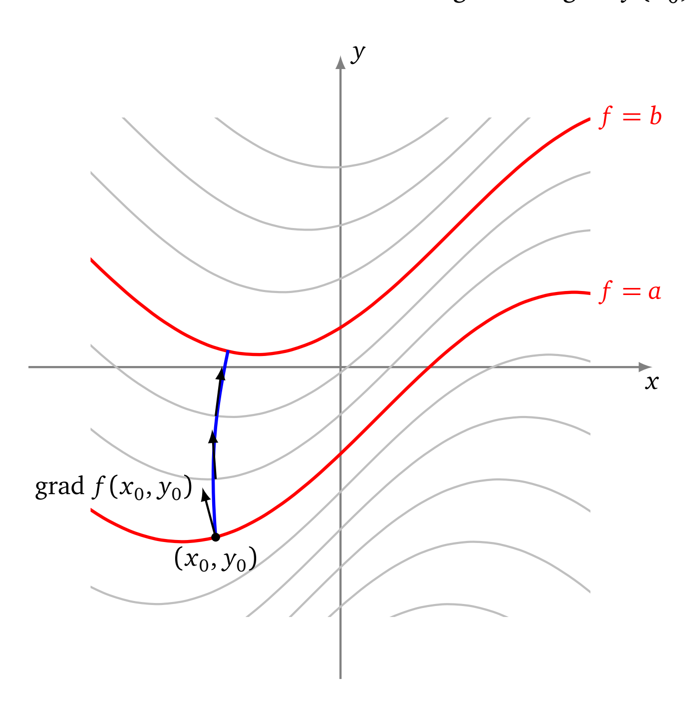
Mais comme $\frac{x_0^2}{a^2}+\frac{y_0^2}{b^2}=1$, l'équation de la tangente se simplifie en $\frac{x_0}{a^2}x+\frac{y_0}{b^2}y=1$.

2.3. Lignes de plus forte pente

Considérons les lignes de niveau $f(x,y)=k$ d'une fonction $f:\mathbb{R}^2\to\mathbb{R}$. On se place en un point (x_0,y_0). On cherche dans quelle direction se déplacer pour augmenter au plus vite la valeur de f.

Proposition 3.
Le vecteur gradient $\operatorname{grad} f(x_0,y_0)$ *indique la direction de plus grande pente à partir du point* (x_0,y_0).

Autrement dit, si l'on veut, à partir d'un point donné (x_0,y_0) de niveau a, passer au niveau $b>a$ le plus vite possible alors il faut démarrer en suivant la direction du gradient $\operatorname{grad} f(x_0,y_0)$.



Comme illustration, un skieur de descente, voulant optimiser sa course, choisira en permanence de s'orienter suivant la plus forte pente, c'est-à-dire dans le sens opposé au gradient.

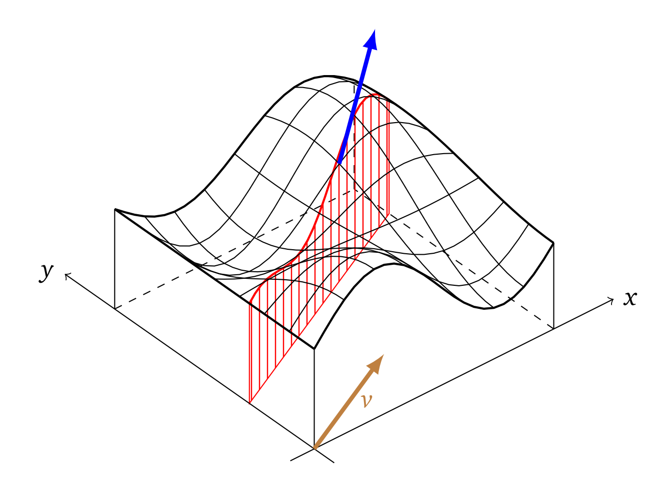
2.4. Dérivée directionnelle

Pour prouver que le gradient indique la ligne de la plus grande pente, nous avons besoin de généraliser la notion de dérivée partielle. Ce passage est plus technique et peut être ignoré en première lecture.

Soit $v=\binom{h}{k}$ un vecteur du plan. La ***dérivée directionnelle*** de f suivant le vecteur v en (x_0,y_0) est le nombre :

$$D_v f(x_0,y_0)=h\frac{\partial f}{\partial x}(x_0,y_0)+k\frac{\partial f}{\partial y}(x_0,y_0).$$

La dérivée directionnelle correspond à la pente de la fonction pour la tranche dirigée par le vecteur v.



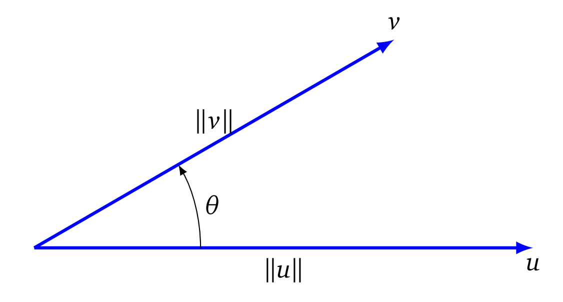
Remarque : pour $v = \binom{1}{0}$ alors $D_v f(x_0, y_0) = \frac{\partial f}{\partial x}(x_0, y_0)$ et pour $v = \binom{0}{1}$ alors $D_v f(x_0, y_0) = \frac{\partial f}{\partial y}(x_0, y_0)$.
On rappelle que le ***produit scalaire*** de deux vecteurs $u = \binom{x}{y}$ et $v = \binom{x'}{y'}$ est donné par

$$\langle u \mid v \rangle = xx' + yy'.$$

On sait que le produit scalaire se calcule aussi géométriquement par :

$$\langle u \mid v \rangle = \|u\| \cdot \|v\| \cdot \cos(\theta)$$

où θ est l'angle entre u et v.



Ainsi, on peut réécrire la dérivée directionnelle sous la forme :

$$D_v f(x_0, y_0) = \langle \operatorname{grad} f(x_0, y_0) \mid v \rangle.$$

On peut maintenant prouver que le gradient indique la ligne de plus grande pente.

Démonstration. La dérivée suivant le vecteur non nul v au point (x_0, y_0) décrit la variation de f autour de ce point lorsqu'on se déplace dans la direction v. La direction selon laquelle la croissance est la plus grande est celle du gradient de f. En effet,

$$D_v f(x_0, y_0) = \langle \operatorname{grad} f(x_0, y_0) \mid v \rangle = \| \operatorname{grad} f(x_0, y_0)\| \cdot \|v\| \cdot \cos\theta$$

où θ est l'angle entre le vecteur $\operatorname{grad} f(x_0, y_0)$ et le vecteur v. Le maximum est atteint lorsque l'angle $\theta = 0$, c'est-à-dire lorsque v pointe dans la même direction que $\operatorname{grad} f(x_0, y_0)$. □

2.5. Surface de niveau

Les résultats présentés ci-dessus pour les fonctions de deux variables se généralisent aux fonctions de trois variables ou plus. Commençons avec trois variables et une fonction $f : \mathbb{R}^3 \to \mathbb{R}$. Rappelons qu'un plan de $\mathbb{R}^3$ passant par (x_0, y_0, z_0) et de vecteur normal $n = (a, b, c)$ a pour équation cartésienne :

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0.$$

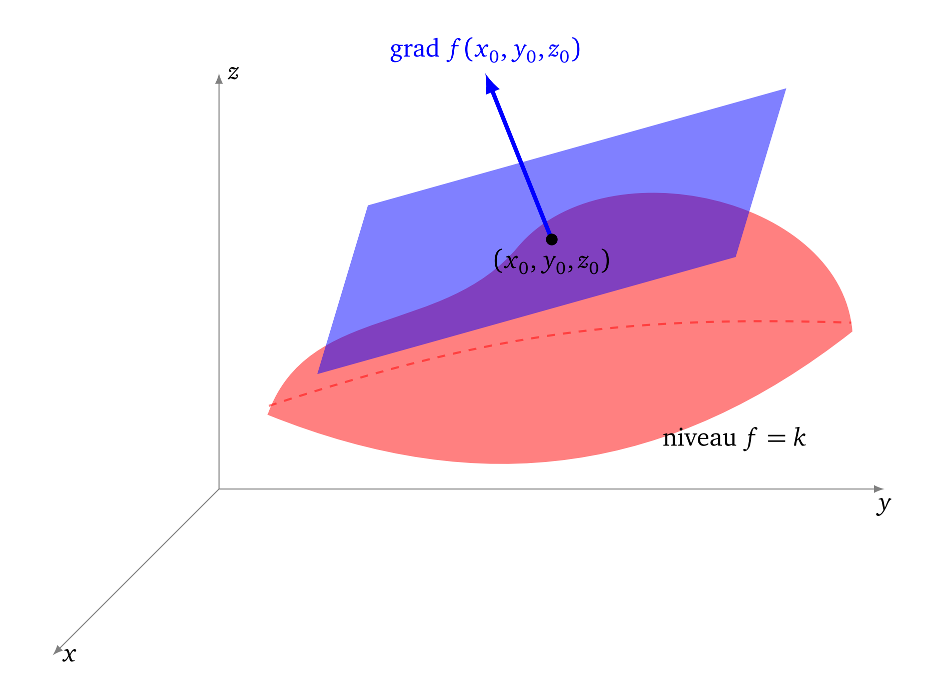
De même qu'il existe une droite tangente pour les lignes de niveau, il existe un ***plan tangent*** à une surface de niveau.

Proposition 4.

Le vecteur gradient $\operatorname{grad} f(x_0, y_0, z_0)$ *est orthogonal à la surface de niveau de* f *passant au point* (x_0, y_0, z_0). *Autrement dit, l'équation du plan tangent à la surface de niveau de* f *en* (x_0, y_0, z_0) *est*

$$\frac{\partial f}{\partial x}(x_0, y_0, z_0)(x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0, z_0)(y - y_0) + \frac{\partial f}{\partial z}(x_0, y_0, z_0)(z - z_0) = 0$$

pourvu que le gradient de f *en ce point ne soit pas le vecteur nul.*



Plus généralement pour $f : \mathbb{R}^n \to \mathbb{R}$, $\operatorname{grad} f(x_1, \ldots, x_n)$ est orthogonal à l'espace tangent à l'hypersurface de niveau $f = k$ passant par le point $(x_1, \ldots, x_n) \in \mathbb{R}^n$ et ce vecteur gradient $\operatorname{grad} f(x_1, \ldots, x_n)$ indique la direction de plus grande pente à partir du point $(x_1, \ldots, x_n)$.

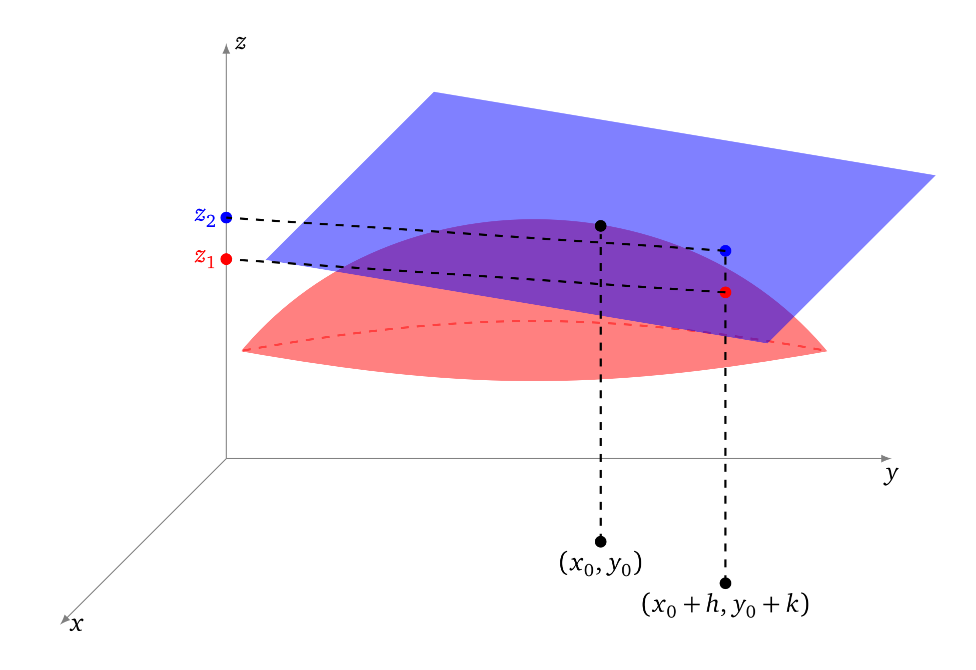
2.6. Calcul approché

Rappelez-vous que la dérivée nous a permis de faire des calculs approchés, par exemple pour estimer $\sqrt{1.01}$ sans calculatrice (voir le chapitre « Dérivée »). Voici, en deux variables, l'analogue de la formule pour une variable :

$$f(x_0 + h, y_0 + k) \simeq f(x_0, y_0) + h\frac{\partial f}{\partial x}(x_0, y_0) + k\frac{\partial f}{\partial y}(x_0, y_0).$$

Cette approximation est valable pour h et k petits.

L'interprétation géométrique est la suivante : on approche le graphe de f en (x_0, y_0) par le plan tangent au graphe en ce point. Sur la figure ci-dessous sont représentés : le graphe de f (en rouge), le plan tangent au-dessus du point (x_0, y_0) (en bleu). La valeur $z_1 = f(x_0+h, y_0+k)$ est la valeur exacte donnée par le point de la surface au dessus de (x_0+h, y_0+k). On approche cette valeur par $z_2 = f(x_0, y_0)+h\frac{\partial f}{\partial x}(x_0, y_0)+k\frac{\partial f}{\partial y}(x_0, y_0)$ donnée par le point du plan tangent au dessus de $(x_0 + h, y_0 + k)$.



Exemple.
Valeur approchée de $f(1.002, 0.997)$ si $f(x,y) = x^2 y$.

Solution. Ici $(x_0, y_0) = (1,1)$, $h = 2 \times 10^{-3}$, $k = -3 \times 10^{-3}$, $\frac{\partial f}{\partial x}(x,y) = 2xy$, $\frac{\partial f}{\partial y}(x,y) = x^2$, donc $\frac{\partial f}{\partial x}(x_0,y_0) = 2$, $\frac{\partial f}{\partial y}(x_0,y_0) = 1$. Ainsi

$$f(1+h, 1+k) \simeq f(1,1) + 2h + k$$

donc

$$f(1.002, 0.997) \simeq 1 + 2 \times 2 \times 10^{-3} - 3 \times 10^{-3} \simeq 1.001.$$

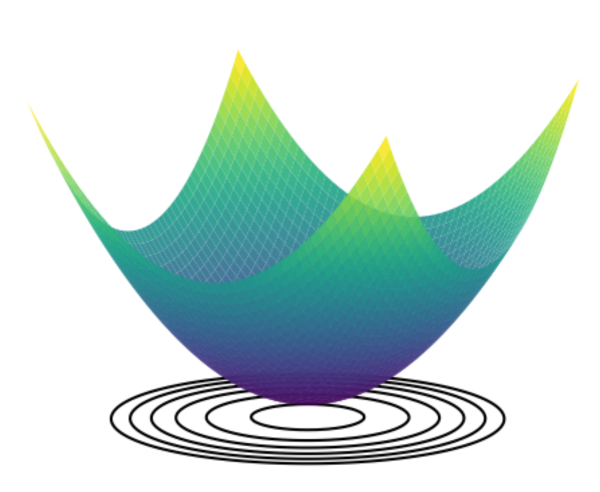
Avec une calculatrice, on trouve $f(1.002, 0.997) = 1.000992$: l'approximation est bonne.

2.7. Minimum et maximum

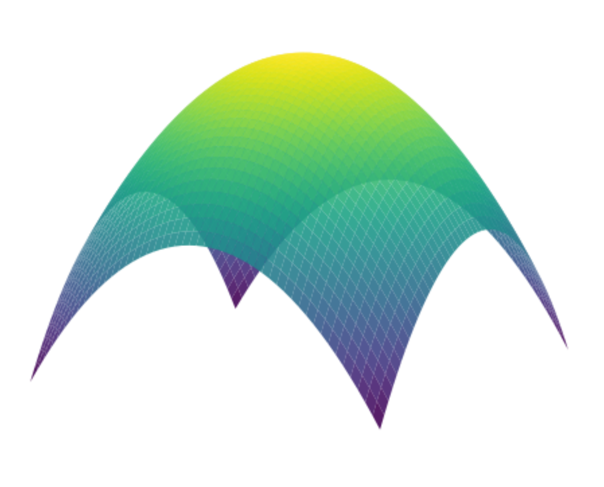
Définition.
Soit $f : \mathbb{R}^2 \to \mathbb{R}$.

- La fonction f admet un ***minimum local*** en (x_0, y_0) s'il existe un disque D centré en ce point tel que
$$f(x,y) \geqslant f(x_0,y_0) \quad \text{pour tout } (x,y) \in D.$$
- La fonction f admet un ***maximum local*** en (x_0, y_0) pour lequel
$$f(x,y) \leqslant f(x_0,y_0) \quad \text{pour tout } (x,y) \in D.$$
- On parle d'un ***extremum local*** pour un minimum ou un maximum local.

Exemple.
L'exemple type de minimum est celui de la fonction $f(x,y) = x^2 + y^2$ en $(0,0)$. Voici son graphe et ses lignes de niveau.



La fonction $f(x,y) = -x^2 - y^2$ admet, elle, un maximum en $(0,0)$.



Proposition 5.
Soit $f : \mathbb{R}^2 \to \mathbb{R}$. Si f admet un minimum ou un maximum local en (x_0, y_0) alors le gradient est le vecteur nul en ce point, autrement dit :

$$\frac{\partial f}{\partial x}(x_0,y_0) = 0 \quad \textit{et} \quad \frac{\partial f}{\partial y}(x_0,y_0) = 0.$$

Démonstration. Prenons le cas d'un minimum local. La fonction d'une variable $x \mapsto f(x, y_0)$ admet aussi un minimum en x_0 donc sa dérivée est nulle en x_0, c'est-à-dire $\frac{\partial f}{\partial x}(x_0, y_0) = 0$. De même $y \mapsto f(x_0, y)$ admet un minimum en y_0 donc $\frac{\partial f}{\partial y}(x_0, y_0) = 0$. □

Dans la suite du cours nous chercherons les points pour lesquels une fonction donnée présente un minimum local. D'après la proposition précédente, ces points sont à chercher parmi les points en lesquels le gradient

s'annule. On dira que (x_0, y_0) est un ***point critique*** de f si les deux dérivées partielles $\frac{\partial f}{\partial x}(x_0, y_0)$ et $\frac{\partial f}{\partial y}(x_0, y_0)$ s'annulent simultanément.

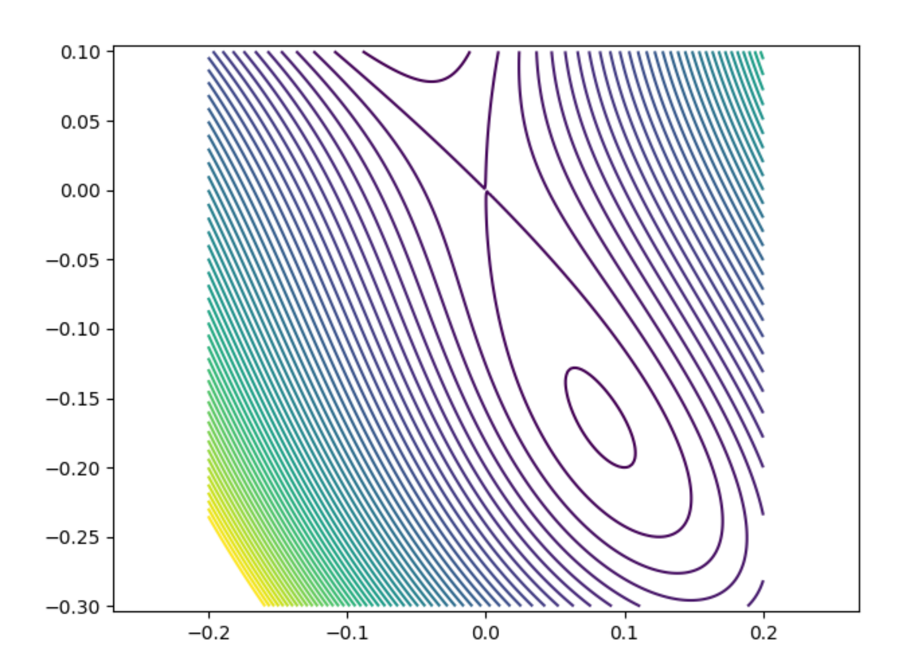
Exemple.
Chercher les points en lesquels $f(x, y) = x^2 - y^3 + xy$ peut atteindre son minimum.
Recherche des points critiques. On calcule

$$\frac{\partial f}{\partial x}(x, y) = 2x + y \quad \text{et} \quad \frac{\partial f}{\partial y}(x, y) = -3y^2 + x.$$

On cherche les points (x, y) en lesquels les deux dérivées partielles s'annulent. Par l'annulation de la première dérivée, on a $2x + y = 0$ donc $y = -2x$. Par l'annulation de la seconde dérivée, on a $-3y^2 + x = 0$ ce qui donne par substitution $-12x^2 + x = 0$, ainsi $x(-12x + 1) = 0$. Donc soit $x = 0$ et alors on a $y = 0$, soit $x = \frac{1}{12}$ et alors $y = -\frac{1}{6}$. Bilan : il y a deux points critiques :

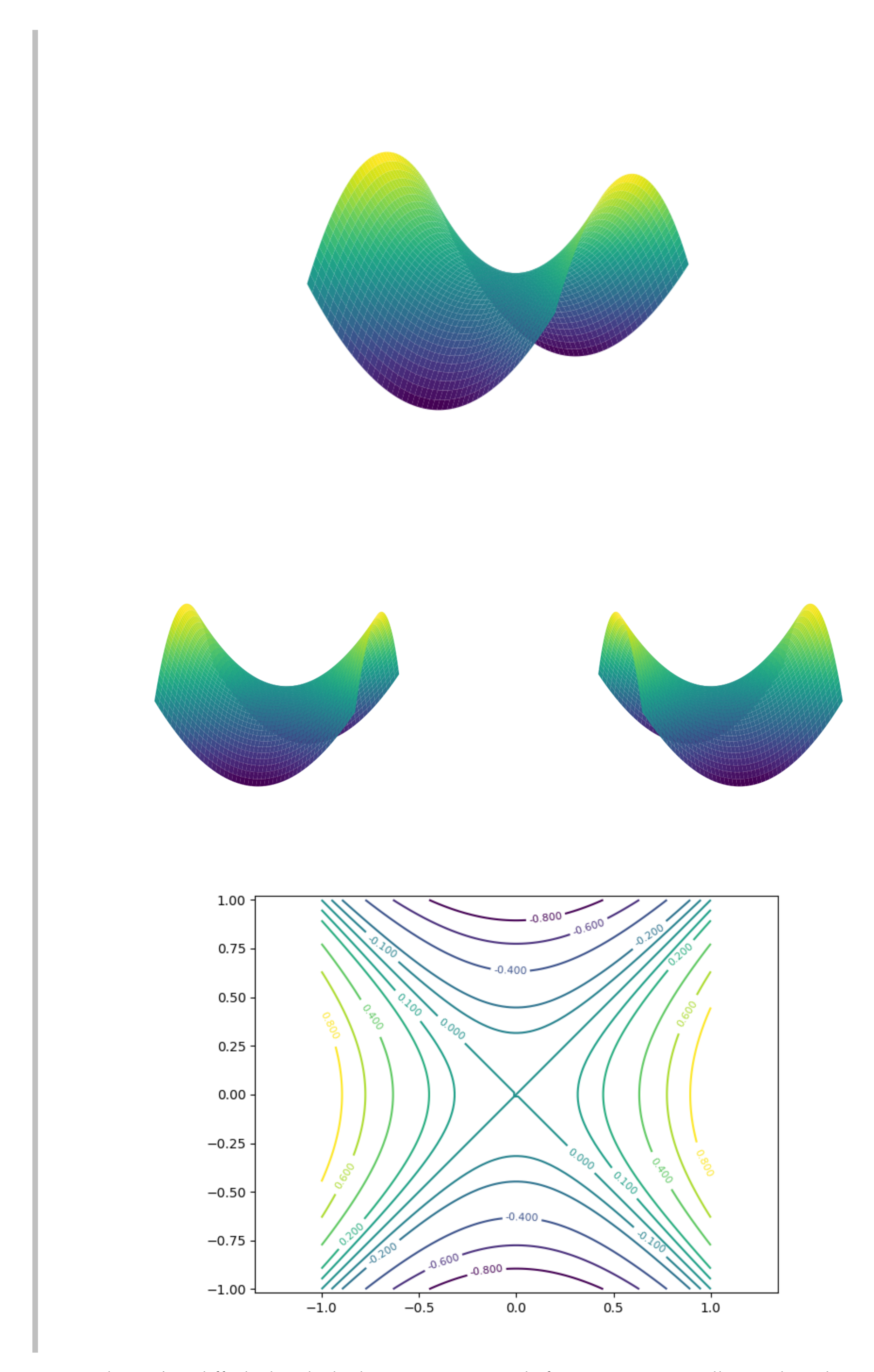
$$(0, 0) \quad \text{et} \quad \left(\frac{1}{12}, -\frac{1}{6}\right).$$

Étude du point critique $(0, 0)$**.** On a $f(0, 0) = 0$ mais on remarque que $f(0, y) = -y^3$ qui peut être négatif ou positif (selon le signe de y proche de 0), donc en $(0, 0)$ il n'y a ni minimum ni maximum.
Étude du point critique $(\frac{1}{12}, -\frac{1}{6})$**.** Il existe un critère (que l'on ne décrira pas ici) qui permet de dire qu'en ce point f admet un minimum local.
Sur le dessin ci-dessous, le minimum est situé à l'intérieur du petit ovale, l'autre point critique en $(0, 0)$ correspond à l'intersection de la ligne de niveau $f = 0$ avec elle-même.



Sur l'exemple précédent, nous avons assez facilement calculé les points critiques à partir des deux équations à deux inconnues. Il faut prendre garde que ce n'est pas un système linéaire et que dans le cas d'une fonction plus compliquée il aurait été impossible de déterminer exactement les points critiques.
On note aussi dans l'exemple précédent que certains points critiques ne sont ni des maximums ni des minimums. L'exemple type, illustré ci-dessous, est celui d'un ***col*** appelé aussi ***point-selle*** en référence à sa forme de selle de cheval.

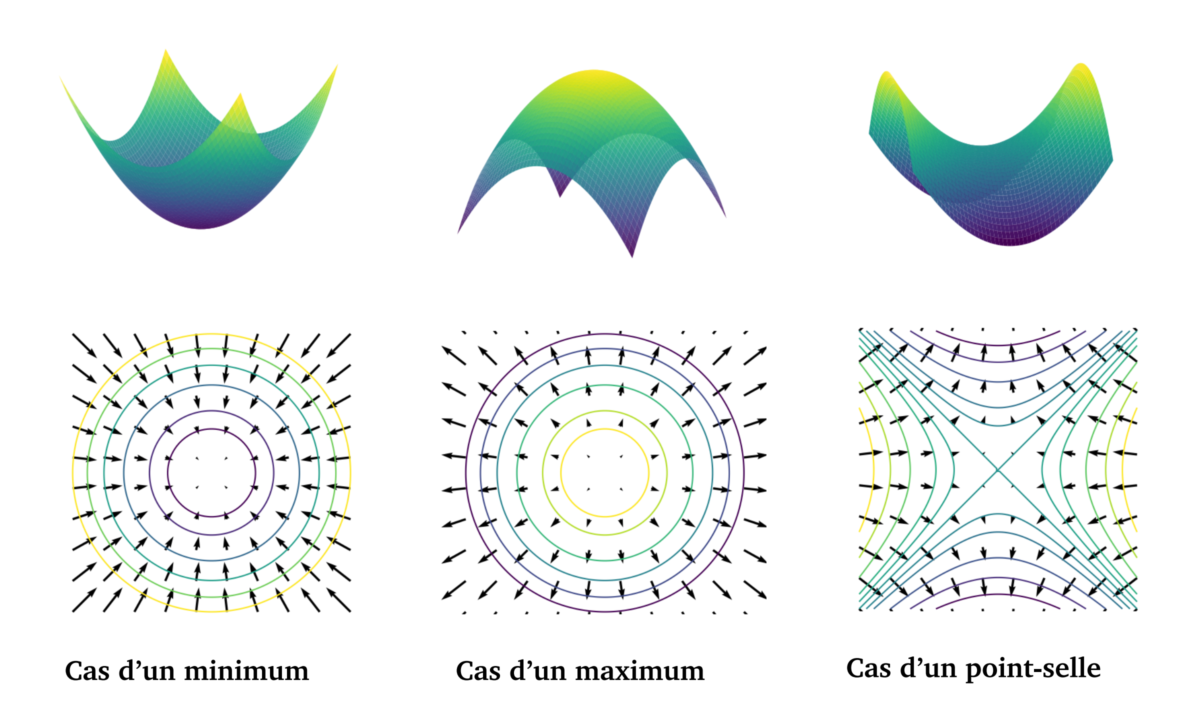
Exemple.
Soit $f(x, y) = x^2 - y^2$. Voici son graphe vu sous trois angles différents et ses lignes de niveau.



Comme il peut être difficile de calculer les points critiques de façon exacte, nous allons utiliser des méthodes

numériques. L'idée qui sera détaillée dans le prochain chapitre est la suivante : comme le gradient indique la direction dans laquelle la fonction f croît le plus rapidement, nous allons suivre la direction opposée au gradient, pour laquelle f décroît le plus rapidement. Ainsi, partant d'un point (x_0, y_0) au hasard, on sait dans quelle direction se déplacer pour obtenir un nouveau point (x_1, y_1) en lequel f est plus petite. Et on recommence.

Sur les trois dessins ci-dessous, on a dessiné les lignes de niveau d'une fonction f ainsi que les vecteurs $-\operatorname{grad} f(x, y)$. On voit que ces vecteurs pointent bien vers le minimum (figure de gauche), s'éloignent d'un maximum (figure centrale), le cas d'un point-selle est spécial (figure de droite). Dans tous les cas, la longueur des vecteurs gradients diminue à l'approche du point critique.



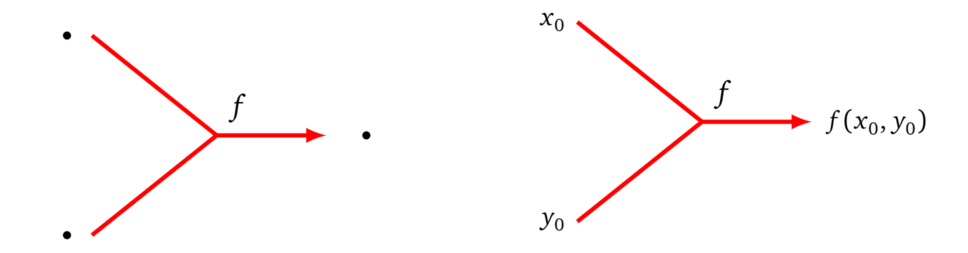
Cas d'un minimum **Cas d'un maximum** **Cas d'un point-selle**

3. Différentiation automatique

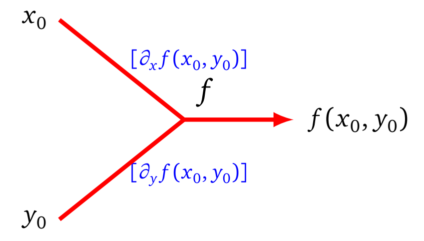
Dans le chapitre « Dérivée », nous avons vu comment calculer la dérivée d'une fonction composée à l'aide de son graphe de calcul. Nous allons faire de même pour les dérivées partielles des fonctions de plusieurs variables afin de calculer le gradient d'une fonction définie par un réseau de neurones.

3.1. Différentiation automatique

Graphe de calcul. Voici le graphe de calcul d'une fonction f de deux variables (schéma de principe à gauche, évaluation en (x_0, y_0) à droite).

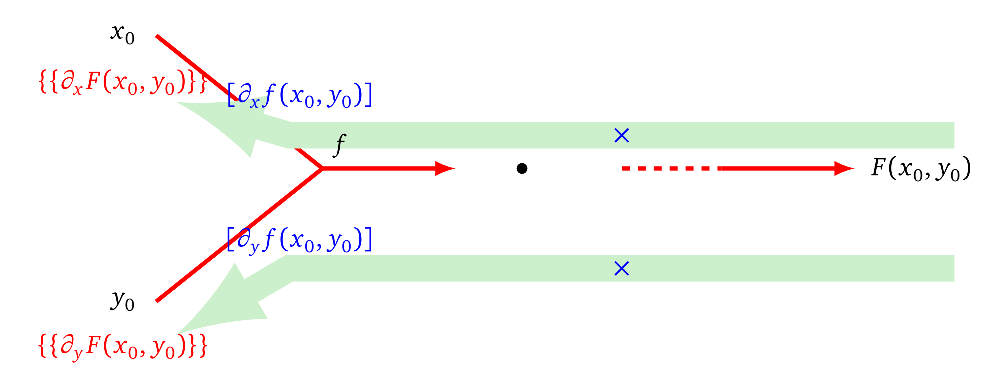


Dérivées locales. Voici la règle pour les dérivées locales à rajouter à chaque branche (entre crochets).

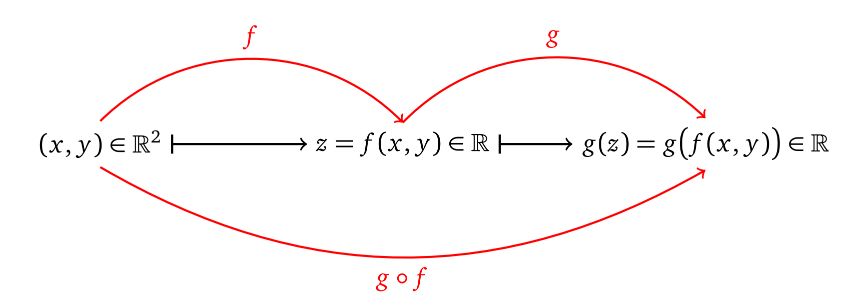


On note $\partial_x f$ comme raccourci de la fonction $\frac{\partial f}{\partial x}$.

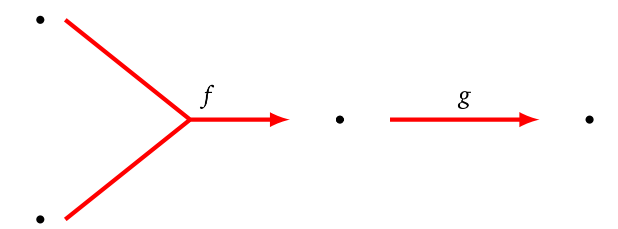
Dérivées partielles. On obtient chacune des dérivées partielles d'une composition F comme le produit des dérivées locales le long des branches allant de la sortie $F(x, y)$ vers l'entrée x (ou y).



Formule mathématique. Soit $f : \mathbb{R}^2 \to \mathbb{R}$ et $g : \mathbb{R} \to \mathbb{R}$.



La composition $F = g \circ f : \mathbb{R}^2 \to \mathbb{R}$ correspond au graphe de calcul dessiné ci-dessous :



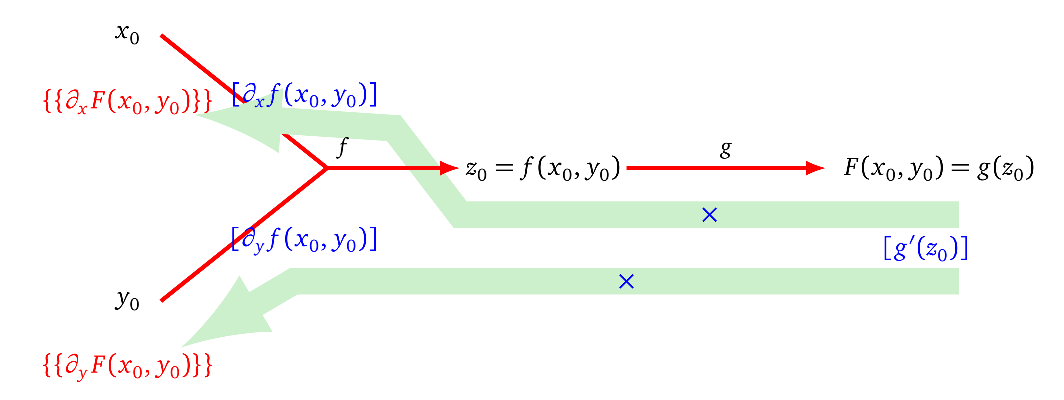
On a donc :

$$F(x, y) = g \circ f(x, y) = g\big(f(x, y)\big).$$

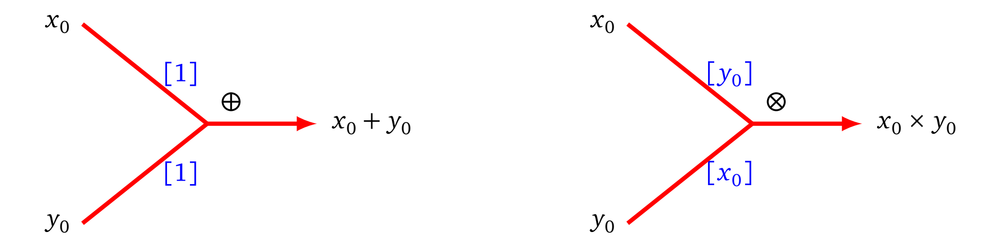
Les dérivées partielles de F sont données par les formules :

$$\boxed{\begin{aligned}\frac{\partial F}{\partial x}(x_0, y_0) &= \frac{\partial f}{\partial x}(x_0, y_0) \cdot g'\big(f(x_0, y_0)\big)\\[1em] \frac{\partial F}{\partial y}(x_0, y_0) &= \frac{\partial f}{\partial y}(x_0, y_0) \cdot g'\big(f(x_0, y_0)\big)\end{aligned}}$$

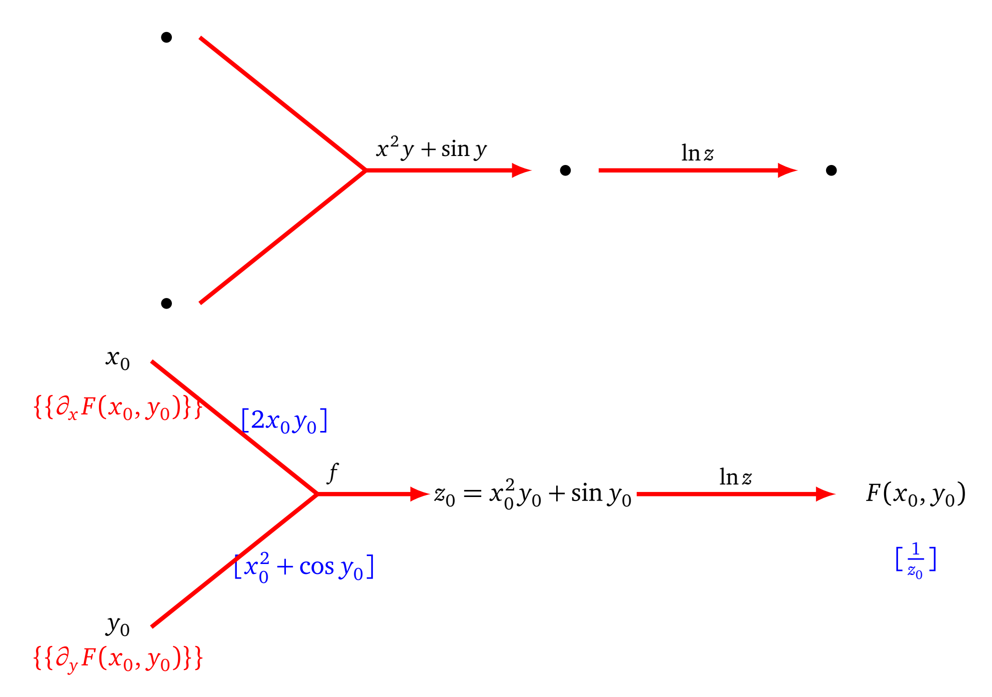
La preuve de la formule pour $\frac{\partial F}{\partial x}(x_0, y_0)$ découle directement de la formule de la dérivée d'une composition pour la fonction d'une seule variable $x \mapsto F(x, y_0)$. Il en est de même pour l'autre dérivée partielle. Ces formules justifient notre règle de calcul : la dérivée partielle est le produit des dérivées locales le long de chacune des branches.



Addition et multiplication.
Dans le cas $f(x, y) = x + y$ et $f(x, y) = x \times y$, on retrouve les dérivées locales déjà utilisées dans le cas d'une seule variable.



Exemple. Soit $F(x, y) = \ln(x^2 y + \sin y)$. On souhaite calculer $\operatorname{grad} F(3, 2)$. Nous allons montrer comment calculer $\operatorname{grad} F(x_0, y_0)$ pour x_0 et y_0 quelconques, puis nous reprendrons les calculs depuis le début dans le cas $(x_0, y_0) = (3, 2)$.



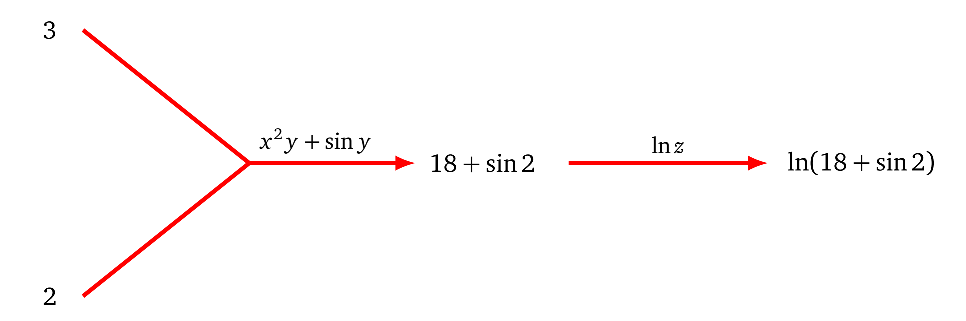
On obtient les dérivées partielles comme produit des dérivées locales :

$$\frac{\partial F}{\partial x}(x_0,y_0) = \left[\frac{1}{z_0}\right] \times [2x_0y_0] = \frac{2x_0y_0}{x_0^2y_0 + \sin y_0},$$

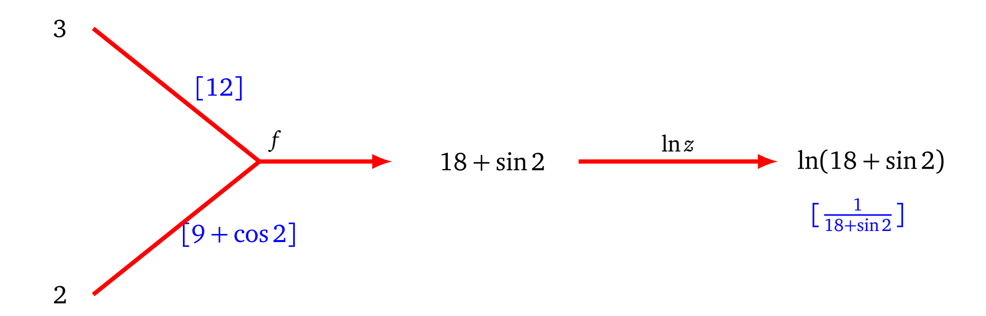
$$\frac{\partial F}{\partial y}(x_0,y_0) = \left[\frac{1}{z_0}\right] \times [x_0^2 + \cos y_0] = \frac{x_0^2 + \cos y_0}{x_0^2y_0 + \sin y_0}.$$

Dans la pratique, pour les réseaux de neurones, on ne calcule jamais l'expression formelle de $\operatorname{grad} F(x_0,y_0)$ mais seulement des gradients en des valeurs (x_0,y_0) données. On reprend donc à chaque fois les étapes ci-dessus mais uniquement pour des valeurs numériques.

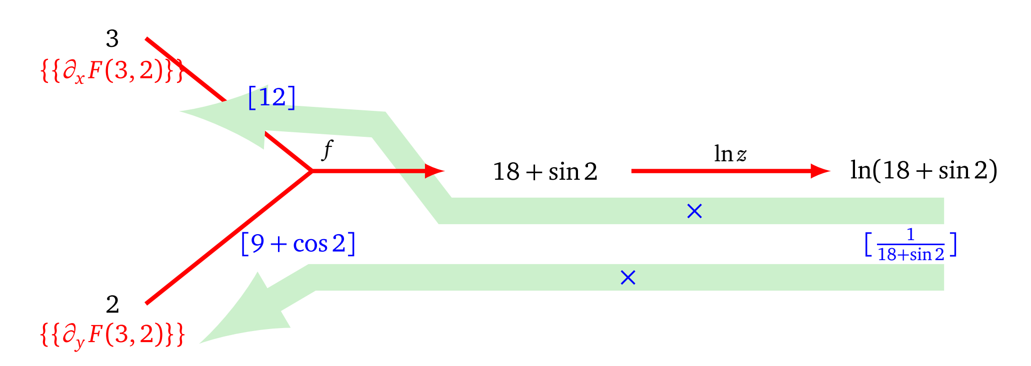
La première étape est de calculer les valeurs des fonctions (de la gauche vers la droite).



La seconde étape est de calculer toutes les dérivées locales. On utilise les valeurs de l'étape précédente et la connaissance des formules de chacune des dérivées des fonctions élémentaires (ici $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ et $\frac{\mathrm{d}\ln}{\mathrm{d}z}$).



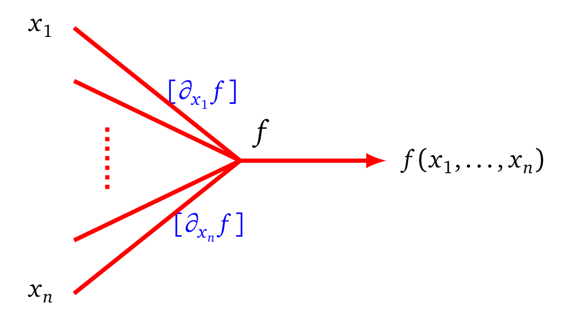
On calcule le produit des dérivées locales le long des arêtes.



On obtient les dérivées partielles :

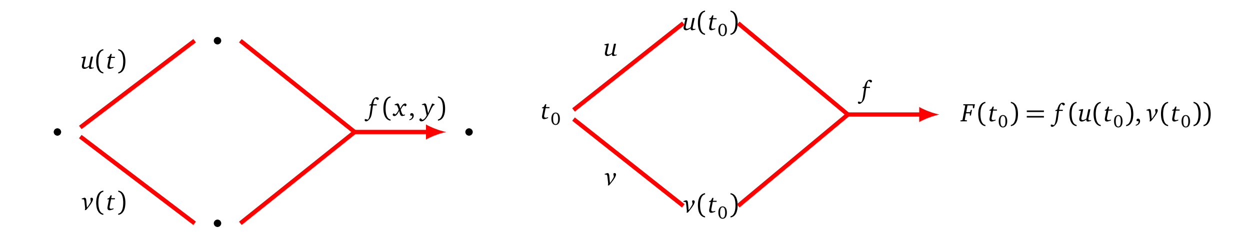
$$\frac{\partial F}{\partial x}(3,2) = \left[\frac{1}{18+\sin 2}\right] \times [12] = \frac{12}{18+\sin 2} \qquad \text{et} \qquad \frac{\partial F}{\partial y}(3,2) = \left[\frac{1}{18+\sin 2}\right] \times [9+\cos 2] = \frac{9+\cos 2}{18+\sin 2}.$$

Règle générale. Dans le cas de n entrées $(x_1,\ldots,x_n)$, la règle des dérivées locales se généralise naturellement : on associe à la branche numéro i la dérivée locale $\frac{\partial f}{\partial x_i}$.

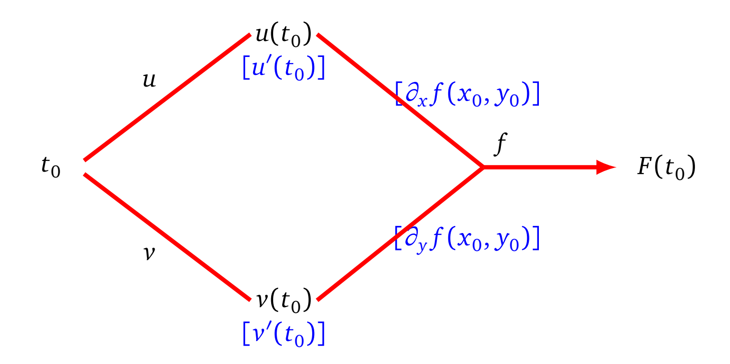


3.2. Différentiation automatique (suite)

Graphe de calcul. Voici le graphe de calcul d'une situation que l'on a déjà rencontrée dans le cas d'une seule variable, mais dont la formule se justifie par les fonctions de deux variables. Il s'agit du graphe de calcul de $F(t) = f\big(u(t), v(t)\big)$ où $u : \mathbb{R} \to \mathbb{R}$, $v : \mathbb{R} \to \mathbb{R}$ et $f : \mathbb{R}^2 \to \mathbb{R}$. L'objectif est de calculer $F'(t)$.

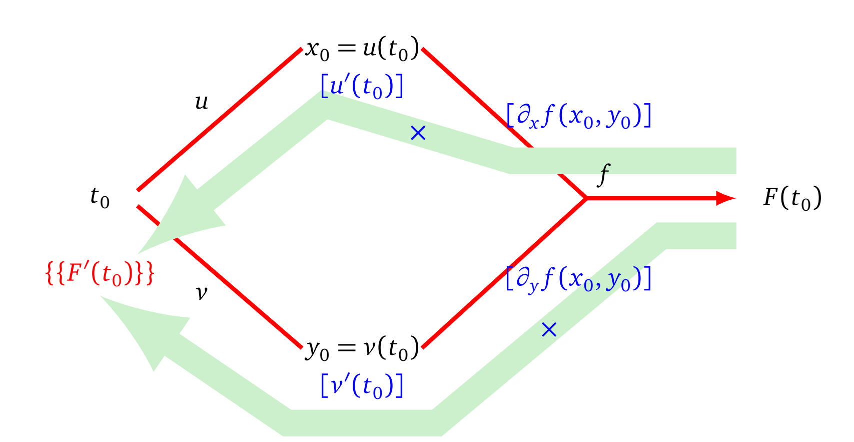


Dérivées locales. On calcule les dérivées locales comme d'habitude.

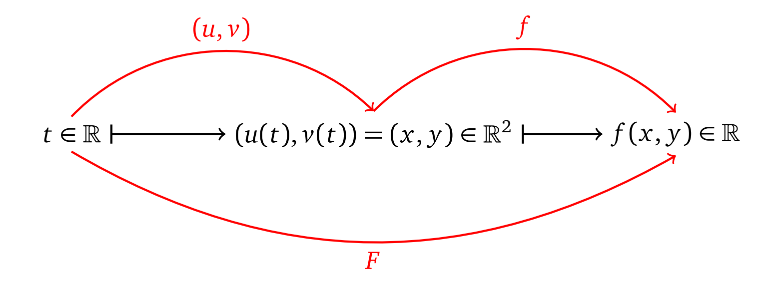


Dérivée. La dérivée s'obtient en deux étapes :

- on calcule le produit des dérivées locales le long des chemins partant de chaque arête sortante jusqu'à la sortie,
- puis on calcule la somme de ces produits.



Formule mathématique. La situation est cette fois la suivante :



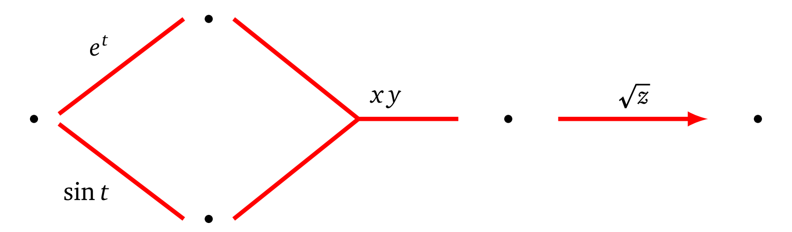
La formule de dérivation de la composition de

$$F(t) = f\big(u(t), v(t)\big)$$

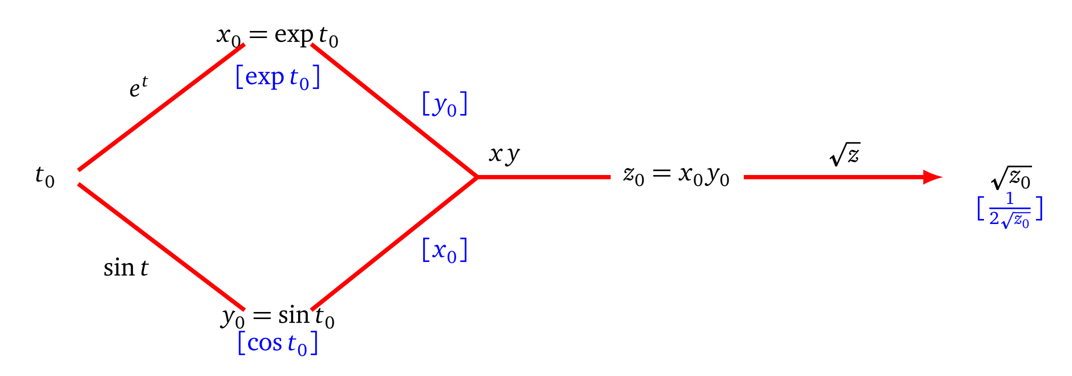
est :

$$\boxed{F'(t_0) = u'(t_0)\frac{\partial f}{\partial x}(u(t_0), v(t_0)) \ + \ v'(t_0)\frac{\partial f}{\partial y}(u(t_0), v(t_0))}$$

Exemple. Soit $F(t) = \sqrt{\exp(t)\sin(t)}$. On souhaite calculer $F'(1)$. On commence par calculer $F'(t_0)$ en général avant de tout reprendre dans la cas $t_0 = 1$. Voici le graphe de calcul :



Une fois complété avec les dérivées locales cela donne :



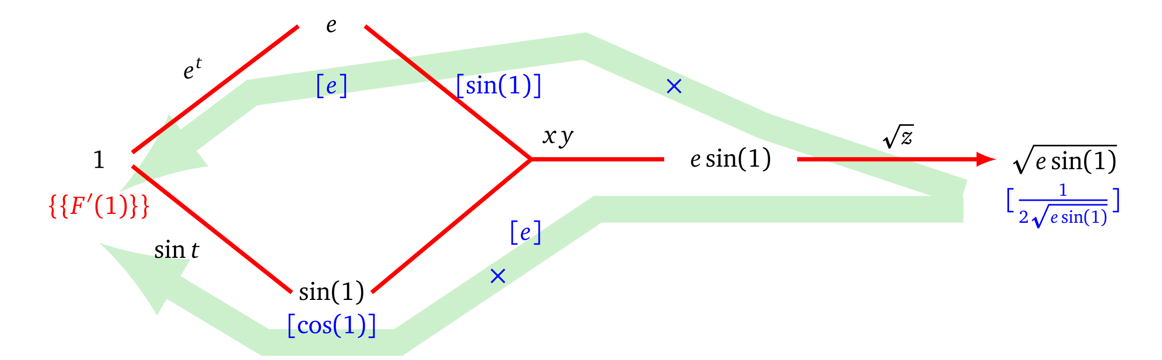
On trouve ainsi :

$$F'(t_0) = \left[\frac{1}{2\sqrt{z_0}}\right] \cdot [y_0] \cdot [\exp t_0] \ + \ \left[\frac{1}{2\sqrt{z_0}}\right] \cdot [x_0] \cdot [\cos t_0]$$

et donc

$$F'(t_0) = \frac{\exp t_0 \cdot (\sin t_0 + \cos t_0)}{2\sqrt{\exp t_0 \sin t_0}}.$$

Reprenons tout depuis le début pour calculer $F'(1)$ en oubliant que l'on a déjà trouvé la formule générale :



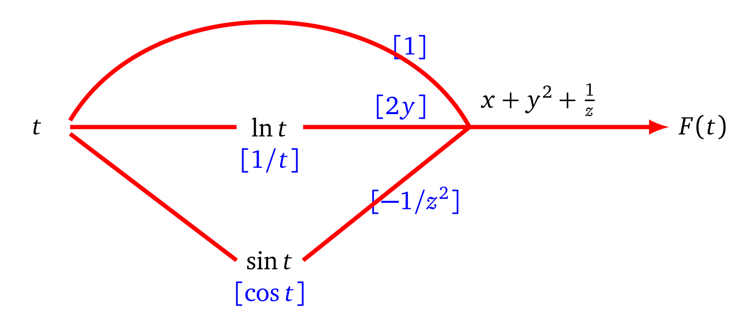
On trouve ainsi :

$$F'(1) = \left[\frac{1}{2\sqrt{e\sin(1)}}\right] \cdot [\sin(1)] \cdot [e] \; + \; \left[\frac{1}{2\sqrt{e\sin(1)}}\right] \cdot [e] \cdot [\cos(1)]$$

et donc

$$F'(1) = \frac{e(\sin(1)+\cos(1))}{2\sqrt{e\sin(1)}}.$$

Règle générale. Dans le cas de n sorties, on somme sur toutes les arêtes sortantes comme dans la situation ci-dessous.

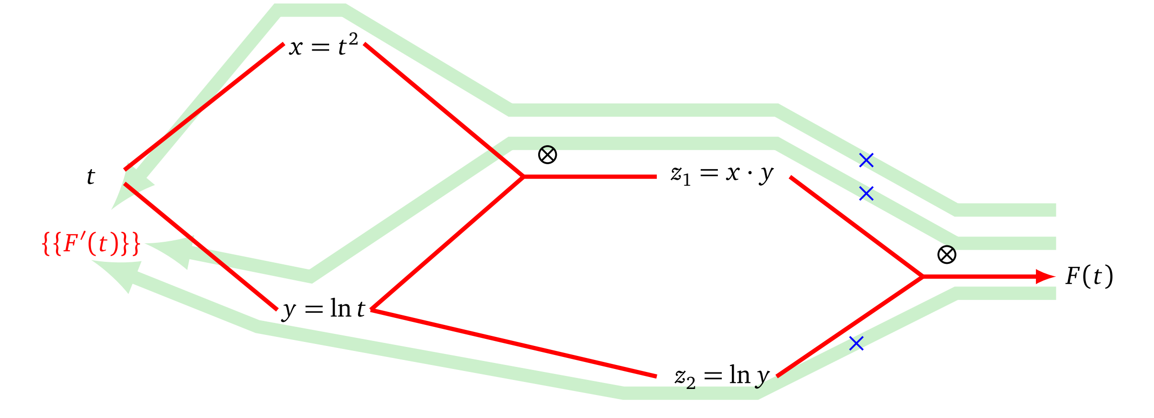


La fonction est $F(t) = t + (\ln t)^2 + \frac{1}{\sin t}$ et en sommant on trouve bien $F'(t) = 1 + \frac{2\ln t}{t} - \frac{\cos t}{\sin^2 t}$.

Un autre exemple. On termine par un exemple plus compliqué : on souhaite calculer la dérivée de

$$F(t) = t^2 \cdot \ln t \cdot \ln(\ln t).$$

Voici le graphe de calcul que l'on utilise (noter qu'avec ce graphe, on ne calcule qu'une seule fois $\ln t$ dont le résultat est réutilisé pour calculer $\ln(\ln t)$).

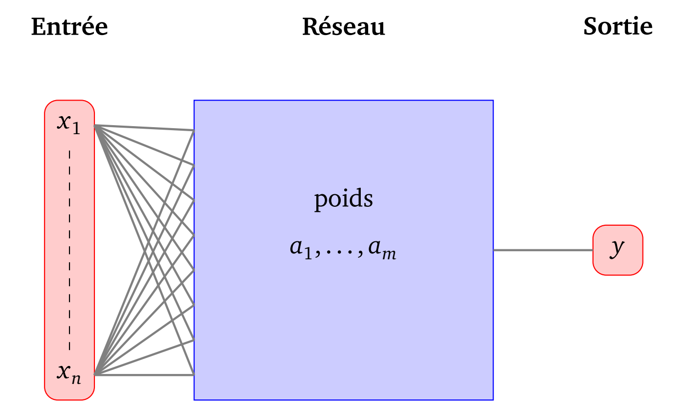


La dérivée s'obtient comme somme sur tous les chemins de la sortie à l'entrée. C'est donc un peu plus compliqué que ce l'on pense au premier abord : il faut ici faire la somme selon trois chemins différents, car l'arête sortant de la variable t vers la variable y se sépare ensuite en deux. Nous allons voir comment gérer cette difficulté dans la section suivante.

4. Gradient pour un réseau de neurones

4.1. Fonction associée à un réseau

Pour un réseau de neurones $\mathcal{R}$ ayant n entrées $(x_1,\dots,x_n)$ et une seule sortie, nous associons une fonction : $F:\mathbb{R}^n \to \mathbb{R}$, $(x_1,\dots,x_n)\mapsto F(x_1,\dots,x_n)$. La situation est en fait plus compliquée. Jusqu'ici les paramètres du réseau étaient donnés. À partir de maintenant les variables du problème ne seront plus les entrées mais les poids du réseau. Notons $a_1,\dots,a_m$ ces poids (l'ensemble des coefficients et des biais). Si l'entrée est fixée et que les poids sont les variables du réseau alors on pourrait considérer que ce même réseau $\mathcal{R}$ définit la fonction $\widetilde{F}:\mathbb{R}^m \to \mathbb{R}$, $(a_1,\dots,a_m)\mapsto \widetilde{F}(a_1,\dots,a_m)$.



Ce dont nous aurons besoin pour la suite et que nous allons calculer dans ce chapitre c'est le gradient de $\widetilde{F}$ par rapport aux poids $(a_1,\dots,a_m)$, autrement dit, il s'agit de calculer :

$$\frac{\partial \widetilde{F}}{\partial a_j}.$$

Pour concilier les deux points de vue (entrées et poids), on dira qu'un réseau de neurones ayant des entrées $(x_1,\dots,x_n)$ et des poids $(a_1,\dots,a_m)$ définit la fonction :

$$\begin{array}{rccc} \widehat{F}: & \mathbb{R}^n\times\mathbb{R}^m & \longrightarrow & \mathbb{R} \\ & (x_1,\dots,x_n),(a_1,\dots,a_m) & \longmapsto & \widehat{F}(x_1,\dots,x_n,a_1,\dots,a_m) \end{array}$$

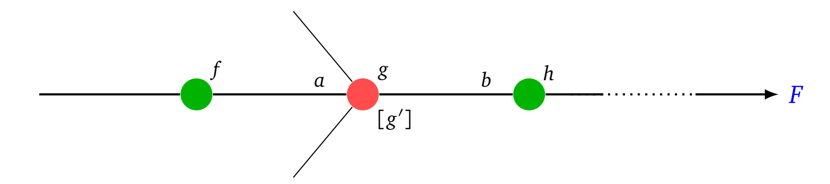
Remarque.
Ce n'est pas tout à fait la fonction $\widetilde{F}$ dont on voudra calculer le gradient mais notre attention se portera sur une fonction d'erreur E de la forme $E=(\widetilde{F}-y_0)^2$, où $\widetilde{F}$ est la fonction définie ci-dessus correspondant à une certaine entrée et à la sortie y_0. On calcule facilement les dérivées partielles de E à partir de celles de $\widetilde{F}$ par la formule :

$$\frac{\partial E}{\partial a_j}=2\frac{\partial \widetilde{F}}{\partial a_j}(\widetilde{F}-y_0).$$

Voir le chapitre « Rétropropagation » pour plus de détails.

4.2. Formule du gradient

On considère la portion suivante d'un réseau de neurones :



On s'intéresse à une seule arête entrante du neurone central rouge, celle qui porte le poids a.

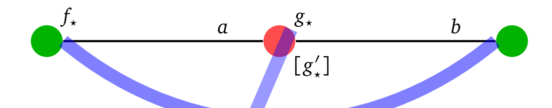
- a et b sont des poids,
- f, g, h sont des fonctions d'activation,
- f', g', h' sont leur dérivées,
- F est la fonction associée au réseau complet.

Pour distinguer la fonction de sa valeur en un point, on notera f la fonction et $f_\star$ la valeur de la fonction à la sortie du neurone correspondant.

Voici la formule pour calculer la dérivée partielle de F par rapport au coefficient a, connaissant la dérivée partielle par rapport au coefficient b.

$$\boxed{\frac{\partial F}{\partial a} = f_\star \cdot \frac{g'_\star}{g_\star} \cdot b \cdot \frac{\partial F}{\partial b}}$$

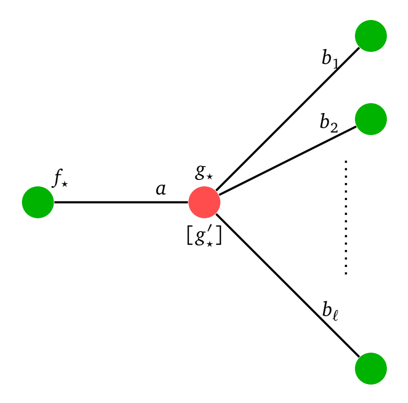
Voici un schéma pour retenir cette « *formule du sourire* » : on multiplie les coefficients $f_\star$, $g'_\star$, b et la dérivée partielle par rapport à b le long de l'arc et on divise par le coefficient $g_\star$ au bout du segment.



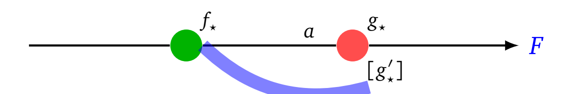
Il est à noter que dans la formule, seule l'arête portant le coefficient a intervient, les autres arêtes entrantes n'interviennent pas (et ne sont pas représentées). Par contre, dans le cas de plusieurs arêtes sortantes il faut calculer la somme des formules précédentes sur chaque arête :

$$\boxed{\frac{\partial F}{\partial a} = \sum_{i=1}^{\ell} f_\star \cdot \frac{g'_\star}{g_\star} \cdot b_i \cdot \frac{\partial F}{\partial b_i}}$$

Cette somme comporte autant de termes que d'arêtes sortantes (il n'y a pas à énumérer tous les chemins entre le sommet et la sortie comme auparavant dans la différentiation automatique).



Pour pouvoir calculer toutes les dérivées partielles, on procède par récurrence, en partant de la fin puis en revenant en arrière de proche en proche.



Voici la formule d'initialisation associée aux coefficients en sortie de réseau, dite « formule du demi-sourire » :

$$\boxed{\frac{\partial F}{\partial a} = f_\star \cdot g'_\star}$$

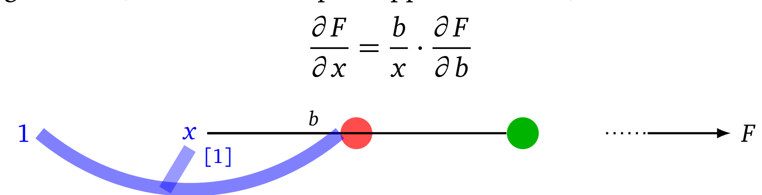
Voici quelques situations particulières, mais qui sont simplement des applications de la formule du sourire.

Formule à l'entrée. On applique la formule du sourire avec x à la place de f.

$$\frac{\partial F}{\partial a} = x \cdot \frac{g_\star'}{g_\star} \cdot b \cdot \frac{\partial F}{\partial b}$$

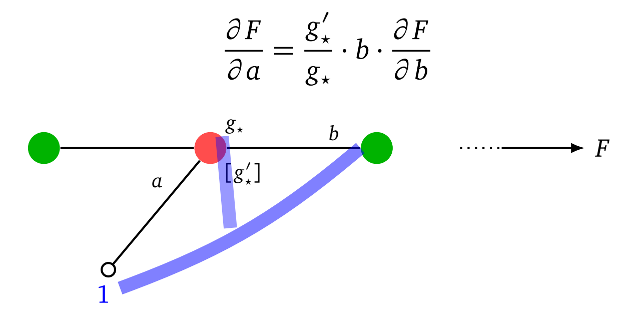
Dérivée partielle par rapport aux variables d'entrées. On applique la formule du sourire en ajoutant des coefficients virtuels égaux à 1 (la dérivée de x par rapport à x est 1) :

$$\frac{\partial F}{\partial x} = \frac{b}{x} \cdot \frac{\partial F}{\partial b}$$



Cas d'un biais. On applique la formule du sourire en ajoutant un coefficient virtuel égal à 1 :

$$\frac{\partial F}{\partial a} = \frac{g_\star'}{g_\star} \cdot b \cdot \frac{\partial F}{\partial b}$$

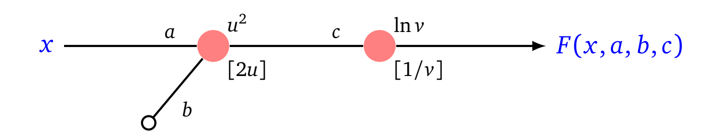


Remarque.

- Ces formules ne sont pas valables lorsque $g_\star = 0$. Nous verrons lors de la preuve de ces formules comment régler ce problème.
- Il faut s'habituer au jeu d'écriture un peu ambigu, comme on le fait pour une expression y qui dépend de x. On peut noter cette expression $y(x)$ ou bien simplement y, cette dernière écriture peut désigner une fonction de x ou bien la valeur prise en x. Par exemple dans la formule du sourire $\frac{\partial F}{\partial a}$ et $\frac{\partial F}{\partial b}$ sont en fait les valeurs des dérivées partielles et ne sont pas vraiment considérées comme des fonctions. Dans la pratique cela ne posera pas de problème car le but est de calculer les valeurs des dérivées partielles (et pas l'expression des fonctions dérivées partielles).
- De plus, les expressions sont dérivées par rapport à différentes variables. Par exemple, si une expression F dépend de y, et l'expression y dépend de x, alors on peut calculer $\frac{\partial F}{\partial y}$, mais aussi $\frac{\partial F}{\partial x}$.
- Enfin la fonction F considérée ici dépend des variables d'entrée x_i mais aussi des poids a_j, si on suivait le formalisme introduit en section 4.1, on devrait plutôt la noter $\widehat{F}$.

4.3. Premier exemple

Voici un réseau très simple.



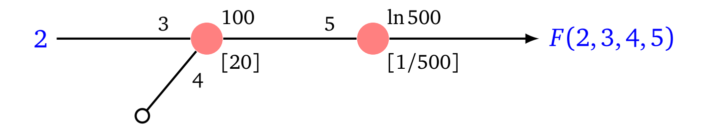
Avec :

- une entrée x, une sortie $F(x)$,
- trois poids a, b, c,
- des fonctions d'activation (plutôt fantaisistes) $u \mapsto u^2$ et $v \mapsto \ln v$.

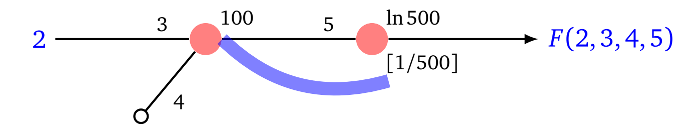
Nous avions l'habitude de considérer la fonction $x \mapsto F(x)$ mais dorénavant les poids sont de nouvelles variables, nous devons donc étudier la fonction $\widehat{F}$ introduite ci-dessus que nous noterons encore F dans la suite :

$$F(x, a, b, c) = \ln\big(c(ax+b)^2\big).$$

Nous souhaitons calculer les dérivées partielles de F par rapport aux poids a, b, c. Nous ne souhaitons pas obtenir une formule générale mais juste la valeur exacte de ces dérivées partielles en un point précis. Nous choisissons l'exemple de $(x, a, b, c) = (2, 3, 4, 5)$. On récrit le réseau avec les valeurs des poids, les valeurs des fonctions et les valeurs des dérivées locales.



Calcul de la dérivée partielle par rapport à c. On part de la sortie pour l'initialisation. On applique la formule du demi-sourire.



$$\frac{\partial F}{\partial c} = 100 \times \frac{1}{500} = \frac{1}{5}.$$

Calcul de la dérivée partielle par rapport à a. On applique la formule du sourire :

$$\frac{\partial F}{\partial a} = 2 \times \frac{20}{100} \times 5 \times \frac{\partial F}{\partial c}.$$

Mais on a déjà calculé $\frac{\partial F}{\partial c} = \frac{1}{5}$ (entre accolades doubles), donc :

$$\frac{\partial F}{\partial a} = \frac{2}{5}.$$

Calcul de la dérivée partielle par rapport à b. On applique la formule du sourire (en posant 1 pour le coefficient manquant) :

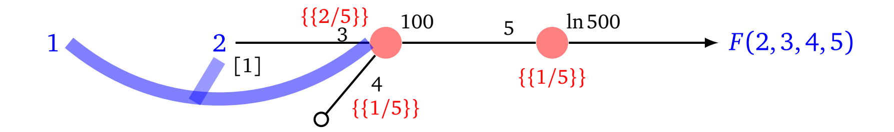
$$\frac{\partial F}{\partial b} = 1 \times \frac{20}{100} \times 5 \times \frac{\partial F}{\partial c}.$$

Donc

$$\frac{\partial F}{\partial b} = \frac{1}{5}.$$

Calcul de la dérivée partielle par rapport à x. On peut aussi calculer cette dérivée partielle, même si nous n'en aurons pas besoin dans les autres chapitres.

$$\frac{\partial F}{\partial x} = 1 \times \frac{1}{2} \times 3 \times \frac{\partial F}{\partial a}.$$



donc

$$\frac{\partial F}{\partial x} = \frac{3}{5}.$$

Bilan. Ainsi $F(2,3,4,5) = \ln 500$ et on a calculé les dérivées partielles (entre doubles accolades) pour chacune des variables x, a, b, c.

Vérification. On peut vérifier nos formules en calculant directement les dérivées partielles à partir de l'expression :

$$F(x,a,b,c) = \ln\big(c(ax+b)^2\big).$$

Par exemple :

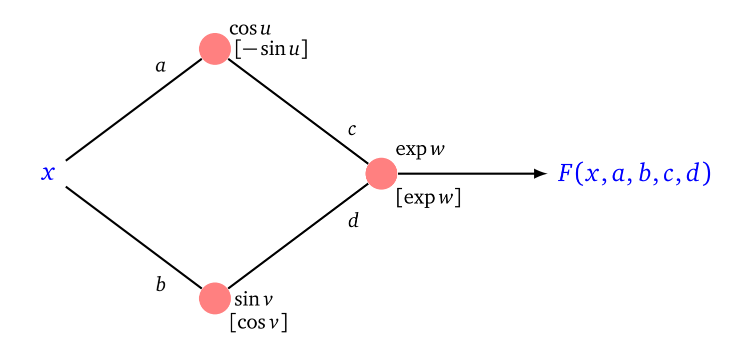
$$\frac{\partial F}{\partial a}(x,a,b,c) = \frac{2cx(ax+b)}{c(ax+b)^2} = \frac{2x}{ax+b}$$

et on a bien

$$\frac{\partial F}{\partial a}(2,3,4,5) = \frac{4}{10} = \frac{2}{5}.$$

4.4. Second exemple

Pour le réseau suivant, on associe comme précédemment une fonction F.



On considère les poids a, b, c, d comme des variables, la fonction $F : \mathbb{R}^5 \to \mathbb{R}$ s'écrit donc :

$$F(x,a,b,c,d) = \exp\big(c\cos(ax) + d\sin(bx)\big).$$

On va noter $u = ax$ et $v = bx$, ainsi $\cos u$ et $\sin v$ sont les sorties des deux premiers neurones. On note $w = c\cos u + d\sin v$. La sortie du troisième neurone (qui est aussi la valeur de F) est alors $\exp w$.

On part de la sortie pour l'initialisation. Il y a deux dérivées partielles à calculer.

Calcul de la dérivée partielle par rapport à c. On applique la formule du demi-sourire :

$$\frac{\partial F}{\partial c} = \cos u \cdot \exp w.$$

Calcul de la dérivée partielle par rapport à d. On applique de nouveau la formule du demi-sourire :

$$\frac{\partial F}{\partial d} = \sin v \cdot \exp w.$$

Calcul de la dérivée partielle par rapport à a. On applique la formule du sourire :

$$\frac{\partial F}{\partial a} = x \cdot \frac{-\sin u}{\cos u} \cdot c \cdot \frac{\partial F}{\partial c}$$

donc

$$\frac{\partial F}{\partial a} = -xc \sin u \exp w.$$

Calcul de la dérivée partielle par rapport à b. On applique la formule du sourire :

$$\frac{\partial F}{\partial b} = x \cdot \frac{\cos v}{\sin v} \cdot d \cdot \frac{\partial F}{\partial d}$$

donc

$$\frac{\partial F}{\partial b} = xd \cos v \exp w.$$

Calcul de la dérivée partielle par rapport à x. Cette dérivée partielle s'obtient comme la somme de deux termes correspondant aux deux arêtes sortantes :

$$\frac{\partial F}{\partial x} = a \cdot \frac{1}{x} \cdot \frac{\partial F}{\partial a} \;+\; b \cdot \frac{1}{x} \cdot \frac{\partial F}{\partial b}$$

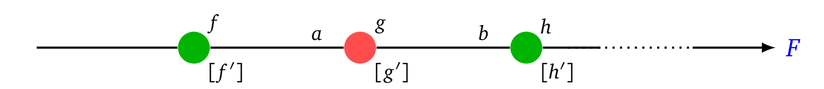
donc

$$\frac{\partial F}{\partial x} = (-ac \sin u + bd \cos v) \exp w.$$

Vérification. En effectuant les substitutions $u = ax$, $v = bx$ et $w = c\cos(ax) + d\sin(bx)$, on retrouve les dérivées partielles attendues, par exemple

$$\frac{\partial F}{\partial x} = \big(-ac\sin(ax) + bd\cos(bx)\big)\exp\big(c\cos(ax) + d\sin(bx)\big).$$

4.5. Preuve et formule générale



Préliminaires.

$$\frac{\partial h}{\partial g} = b \cdot h'_\star \qquad (1)$$

Preuve : on a $h_\star = h(bg_\star)$, la formule s'obtient en dérivant $g \mapsto h(bg)$ par rapport à la variable g, avec $h'_\star = h'(bg_\star)$.

$$\frac{\partial g}{\partial a} = f_\star \cdot g'_\star \qquad (2)$$

Preuve : on a $g_\star = g(\cdots + af_\star + \cdots)$, la formule s'obtient en dérivant $a \mapsto g(\cdots + af + \cdots)$ par rapport à la variable a. On note $g'_\star = g'(\cdots + af_\star + \cdots)$.

$$\frac{\partial h}{\partial b} = g_\star \cdot h'_\star \qquad (3)$$

Preuve : c'est la même formule que l'équation (2) mais cette fois pour $b \mapsto h(bg)$ et $h'_\star = h'(bg_\star)$.

Formule générale.

$$\frac{\partial F}{\partial a} = \frac{\partial F}{\partial g} \cdot f_\star \cdot g'_\star \qquad (4)$$

Preuve : c'est la formule
$$\frac{\partial F}{\partial a} = \frac{\partial F}{\partial g} \cdot \frac{\partial g}{\partial a}$$
(valable car g est une fonction d'une seule variable) suivie de l'application de l'équation (2).

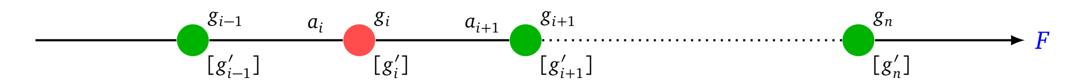
$$\boxed{\frac{\partial F}{\partial g} = \frac{\partial F}{\partial h} \cdot b \cdot h'_\star} \tag{5}$$

Preuve : c'est la formule
$$\frac{\partial F}{\partial g} = \frac{\partial F}{\partial h} \cdot \frac{\partial h}{\partial g}$$
suivie de l'application de l'équation (1).
Dans le cas d'un neurone de sortie on a :

$$\boxed{\frac{\partial F}{\partial g} = 1} \tag{6}$$

car comme g est la dernière fonction, on a $F_\star = g_\star$.

Algorithme.



Voici comment calculer toutes les dérivées partielles voulues (y compris dans le cas $g_\star = 0$ qui avait été exclu dans la formule du sourire).

- On part du neurone de sortie pour lequel on initialise le processus par la formule (6) ce qui donne $\frac{\partial F}{\partial g_n} = 1$.
- On procède par récurrence à rebours. On suppose que l'on a déjà calculé $\frac{\partial F}{\partial g_{i+1}}$ On en déduit :
$$\frac{\partial F}{\partial g_i} = \frac{\partial F}{\partial g_{i+1}} \cdot a_{i+1} \cdot g'_{i+1,\star}$$
par la formule (5).
- Cela permet de calculer les dérivées partielles par rapport aux poids à l'aide de la formule (4) :
$$\frac{\partial F}{\partial a_i} = \frac{\partial F}{\partial g_i} \cdot g_{i-1,\star} \cdot g'_{i,\star}.$$

Preuve de la formule du sourire.
On va exprimer $\frac{\partial F}{\partial a}$ directement en fonction de $\frac{\partial F}{\partial b}$.
Par les équations (4) et (5), on a d'une part :
$$\frac{\partial F}{\partial a} = \frac{\partial F}{\partial h} \cdot b \cdot h'_\star \cdot f_\star \cdot g'_\star \tag{7}$$
et d'autre part :
$$\frac{\partial F}{\partial b} = \frac{\partial F}{\partial h} \cdot \frac{\partial h}{\partial b}.$$
Donc, en utilisant l'équation (3), on obtient :
$$\frac{\partial F}{\partial b} = \frac{\partial F}{\partial h} \cdot g_\star \cdot h'_\star. \tag{8}$$
Cette dernière équation permet de calculer $\frac{\partial F}{\partial h}$ en fonction $\frac{\partial F}{\partial b}$. Ainsi des équations (7) et (8), on obtient la formule du sourire :
$$\frac{\partial F}{\partial a} = f_\star \cdot \frac{g'_\star}{g_\star} \cdot b \cdot \frac{\partial F}{\partial b}.$$
Dans le cas de plusieurs arêtes sortantes, il s'agit de faire une somme comme on l'a déjà vu lors de la différentiation automatique.

Chapitre 8

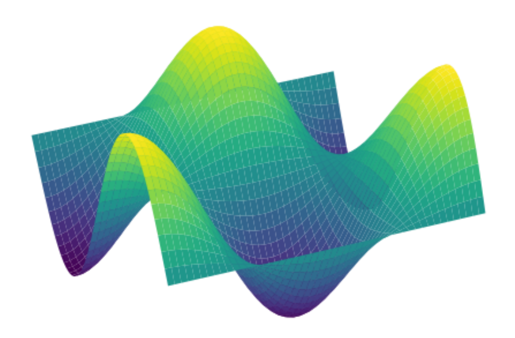
Descente de gradient

Vidéo ■ partie 8.1. Descente de gradient classique
Vidéo ■ partie 8.2. Descente de gradients : exemples
Vidéo ■ partie 8.3. Descente de gradient et régression linéaire
Vidéo ■ partie 8.4. Descente de gradient stochastique
Vidéo ■ partie 8.5. Accélération de la descente de gradient

L'objectif de la méthode de descente de gradient est de trouver un minimum d'une fonction de plusieurs variables le plus rapidement possible. L'idée est très simple, on sait que le vecteur opposé au gradient indique une direction vers des plus petites valeurs de la fonction, il suffit donc de suivre d'un pas cette direction et de recommencer. Cependant, afin d'être encore plus rapide, il est possible d'ajouter plusieurs paramètres qui demandent pas mal d'ingénierie pour être bien choisis.

1. Descente de gradient classique

Imaginons une goutte d'eau en haut d'une colline. La goutte d'eau descend en suivant la ligne de plus grande pente et elle s'arrête lorsqu'elle atteint un point bas. C'est exactement ce que fait la descente de gradient : partant d'un point sur une surface, on cherche la pente la plus grande en calculant le gradient et on descend d'un petit pas, on recommence à partir du nouveau point jusqu'à atteindre un minimum local.



1.1. Où est le minimum ?

On nous donne une fonction f de deux variables (a, b) et nous cherchons un point $(a_{\min}, b_{\min})$ en lequel f atteint un minimum. Voici la méthode expliquée par des dessins sur lesquels ont été tracées des lignes de niveau.

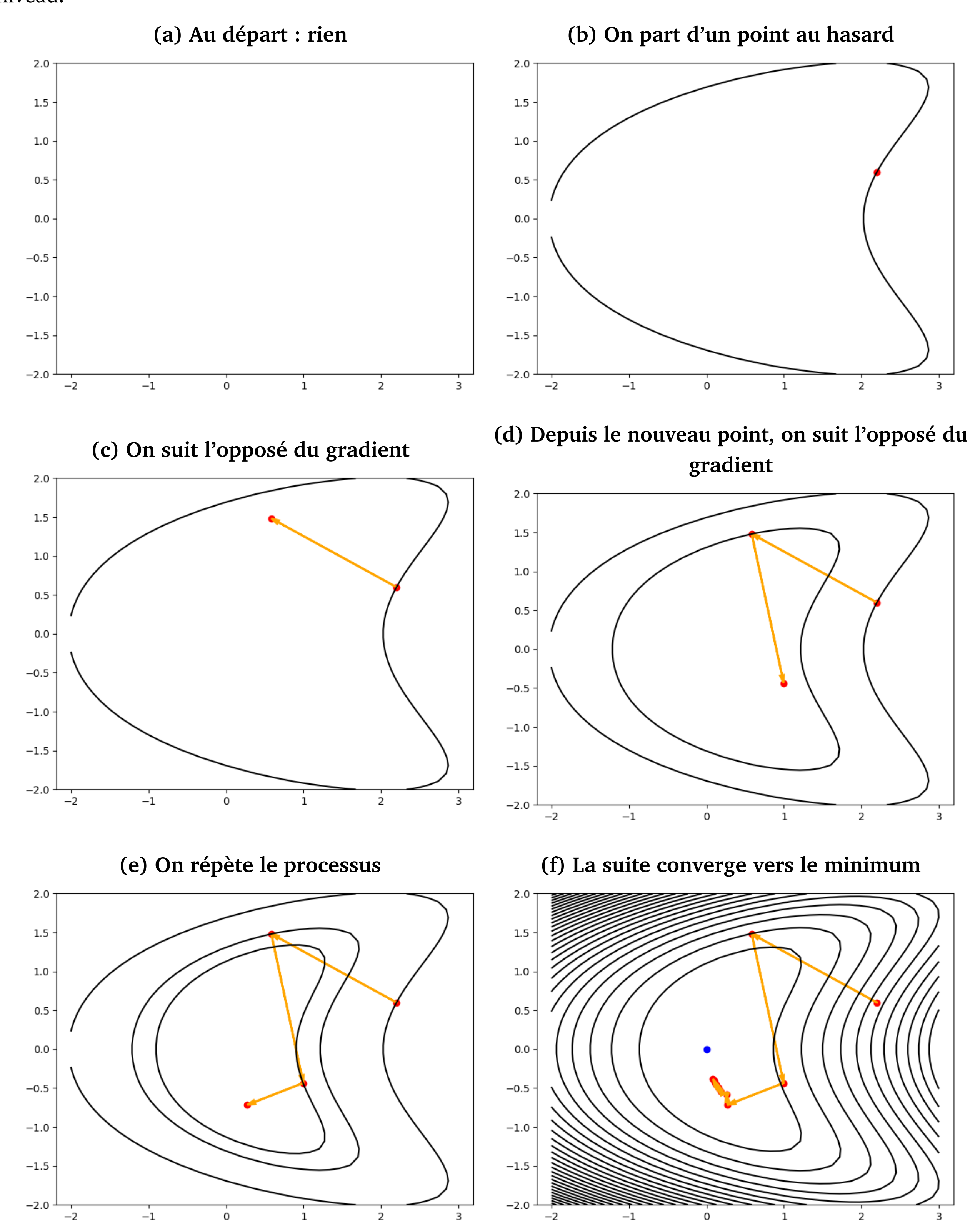
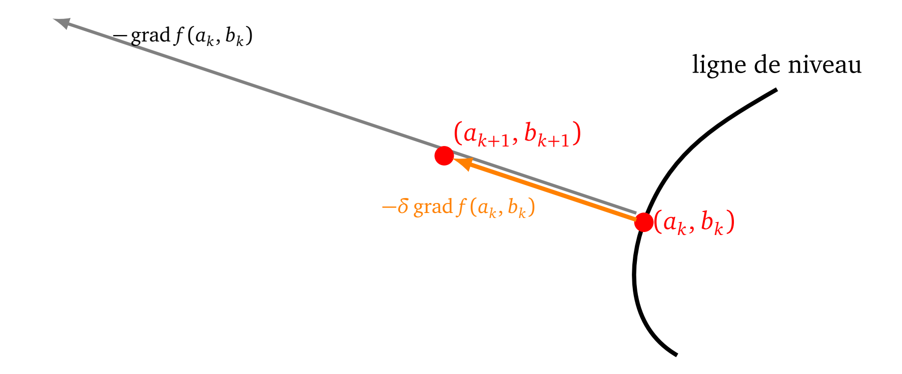


Figure (a). Au départ nous n'avons aucune information globale sur f. La seule opération que l'on s'autorise c'est calculer grad $f(a, b)$ en certains points.

Figure (b). On choisit un point (a_0, b_0) au hasard. Si on note $c_0 = f(a_0, b_0)$ la valeur de f en ce point, on sait que la ligne de niveau $(f = c_0)$ passe par (a_0, b_0).

Figure (c). On calcule en ce point le gradient de f. On trace l'opposé du gradient : $-\operatorname{grad} f(a_0, b_0)$. On sait d'une part que la ligne de niveau est orthogonale à ce gradient et surtout que dans la direction de $-\operatorname{grad} f(a_0, b_0)$, les valeurs de f vont diminuer.



On se dirige alors dans la direction opposée au gradient d'un facteur δ (par exemple $\delta = 0.1$). On arrive à un point noté (a_1, b_1). Par construction, si δ est assez petit, la valeur $c_1 = f(a_1, b_1)$ est plus petite que c_0.
Figure (d). On recommence depuis (a_1, b_1). On calcule l'opposé du gradient en (a_1, b_1), on se dirige dans cette nouvelle direction pour obtenir un point (a_2, b_2) où $c_2 = f(a_2, b_2) < c_1$.
Figure (e). On itère le processus pour obtenir une suite de points (a_k, b_k) pour lesquels f prend des valeurs de plus en plus petites.
Figure (f). On choisit de s'arrêter (selon une condition préalablement établie) et on obtient une valeur approchée (a_N, b_N) du point $(a_{\min}, b_{\min})$ en lequel f atteint son minimum.
Évidemment avec la vision globale de la fonction, on se dit qu'on aurait pu choisir un point de départ plus près et que certaines directions choisies ne sont pas les meilleures. Mais souvenez-vous que l'algorithme est « aveugle », il ne calcule pas les valeurs de f en les (a_k, b_k) et n'a pas connaissance du comportement de f au voisinage de ces points.

1.2. Exemple en deux variables

Prenons l'exemple de $f(a, b) = a^2 + 3b^2$ dont le minimum est bien évidemment atteint en $(0, 0)$ et appliquons la méthode du gradient.
Nous aurons besoin de calculer la valeur du gradient en certains points par la formule :

$$\operatorname{grad} f(a, b) = \left(\frac{\partial f}{\partial a}(a, b), \frac{\partial f}{\partial b}(a, b)\right) = (2a, 6b).$$

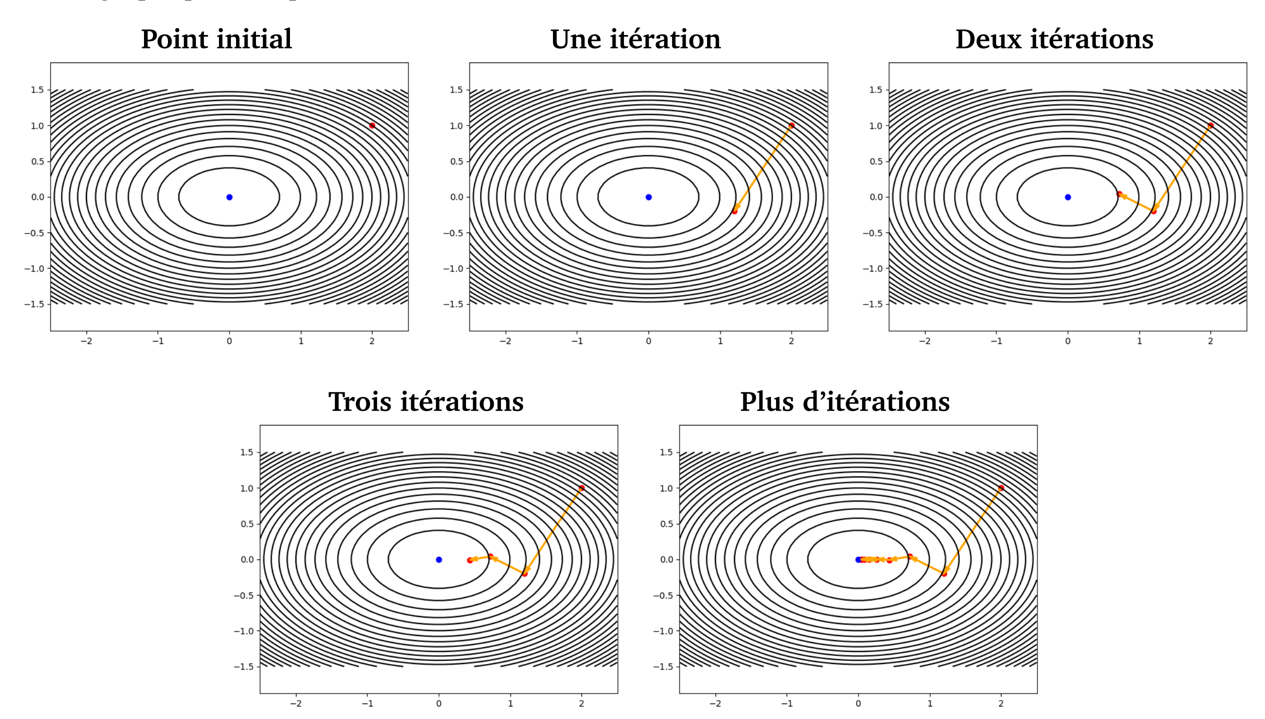
Tout d'abord, on part d'un point $(a_0, b_0) = (2, 1)$ par exemple. Même si nous n'en avons pas besoin pour notre construction, on a $f(a_0, b_0) = 7$. On calcule $\operatorname{grad} f(a_0, b_0) = (4, 6)$. On fixe le facteur $\delta = 0.2$. On se déplace dans la direction opposée à ce gradient :

$$(a_1, b_1) = (a_0, b_0) - \delta \operatorname{grad} f(a_0, b_0) = (2, 1) - 0.2(4, 6) = (2, 1) - (0.8, 1.2) = (1.2, -0.2).$$

On note que $f(a_1, b_1) = 1.56$ est bien plus petit que $f(a_0, b_0)$. On recommence ensuite depuis (a_1, b_1). En quelques étapes les valeurs de f tendent vers la valeur minimale et, dans notre cas, la suite converge vers $(0, 0)$ (les valeurs sont approchées).

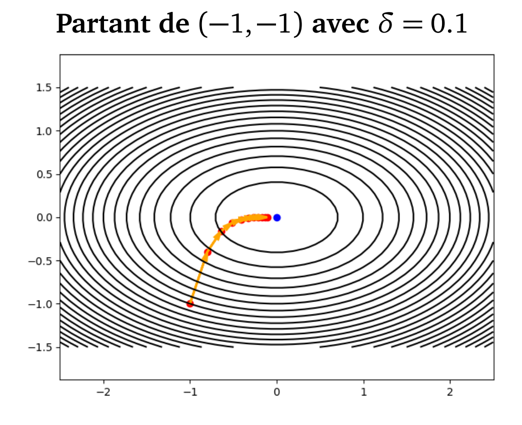
k	(a_k, b_k)	$\operatorname{grad} f(a_k, b_k)$	$f(a_k, b_k)$
0	$(2, 1)$	$(4, 6)$	7
1	$(1.2, -0.2)$	$(2.4, -1.20)$	1.56
2	$(0.72, 0.04)$	$(1.44, 0.24)$	0.523
3	$(0.432, -0.008)$	$(0.864, -0.048)$	0.186
4	$(0.2592, 0.0016)$	$(0.5184, 0.0096)$	0.067
5	$(0.15552, -0.00032)$	$(0.31104, -0.00192)$	0.024
...			
10	$(0.012, 1.02 \cdot 10^{-7})$	$(0.024, 6.14 \cdot 10^{-7})$	0.00014
...			
20	$(7.31 \cdot 10^{-5}, 1.04 \cdot 10^{-14})$	$(1.46 \cdot 10^{-4}, 6.29 \cdot 10^{-14})$	$5.34 \cdot 10^{-9}$

Voici les graphiques des premières itérations :



Que se passe-t-il si l'on part d'un autre point ? Partons cette fois de $(a_0, b_0) = (-1, -1)$ et fixons le pas à $\delta = 0.1$. Alors $(a_1, b_1) = (-0.8, -0.4)$, $(a_2, b_2) = (-0.64, -0.16)$... La suite converge également vers $(0, 0)$.

Partant de $(-1, -1)$ avec $\delta = 0.1$



1.3. Exemples en une variable

La descente de gradient fonctionne aussi très bien pour les fonctions d'une seule variable et sa visualisation est instructive.

Exemple.
Considérons la fonction $f : \mathbb{R} \to \mathbb{R}$ définie par

$$f(a) = a^2 + 1.$$

Il s'agit de trouver la valeur en laquelle f atteint son minimum, c'est clairement $a_{\min} = 0$ pour lequel $f(a_{\min}) = 1$. Retrouvons ceci par la descente de gradient.
Partant d'une valeur a_0 quelconque, la formule de récurrence est :

$$a_{k+1} = a_k - \delta \operatorname{grad} f(a)$$

où δ est le pas, choisi assez petit, et $\operatorname{grad} f(a) = f'(a) = 2a$. Autrement dit :

$$a_{k+1} = a_k - 2\delta a_k.$$

Voici le tableau des valeurs pour un pas $\delta = 0.2$ et une valeur initiale $a_0 = 2$.

k	a_k	$f'(a_k) = \operatorname{grad} f(a_k)$	$f(a_k)$
0	2	4	5
1	1.2	2.4	2.44
2	0.72	1.44	1.5184
3	0.43	0.86	1.1866
4	0.25	0.5184	1.0671
5	0.15	0.31	1.0241
6	0.093	0.186	1.0087
7	0.055	0.111	1.0031
8	0.033	0.067	1.0011
9	0.020	0.040	1.0004
10	0.012	0.024	1.0001

Voici la version graphique de ces 10 premières itérations (figure de gauche). Si l'on change le point initial, ($a_0 = -1.5$ sur la figure de droite) alors la suite (a_k) converge vers la même valeur $a_{\min} = 0$.

$\delta = 0.2 \quad a_0 = 2$

$\delta = 0.2 \quad a_0 = -1.5$

Il faut bien comprendre ce graphique : la suite des points (a_k) (points rouges) se lit sur l'axe des abscisses. Les vecteurs (orange) montrent les itérations. Il est plus facile de comprendre l'algorithme sur le graphe de f (en bleu). Sur ce graphe, on reporte les points $(a_k, f(a_k))$ (en rouge), ce qui permet de bien comprendre

que les valeurs $f(a_k)$ décroissent rapidement. On note aussi que le gradient (ici $f'(a_k)$) diminue à l'approche du minimum, ce qui se traduit par des vecteurs (c'est-à-dire l'écart entre deux points successifs) de plus en plus petits.

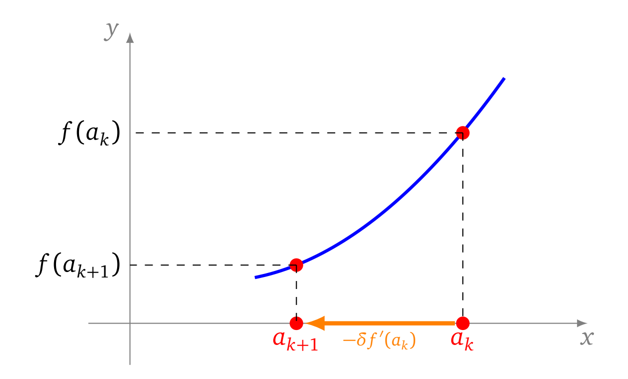
Justifions l'algorithme et l'intervention du gradient dans le cas d'une variable. Si la fonction est croissante sur un intervalle, $f'(a) > 0$ pour tout a dans cet intervalle et la formule

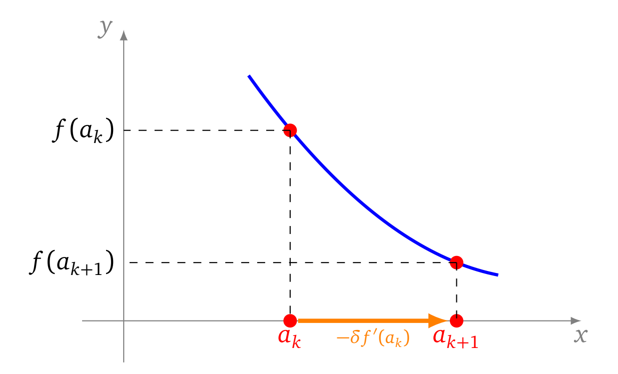
$$a_{k+1} = a_k - \delta f'(a_k) \quad \text{donne} \quad a_{k+1} < a_k.$$

Ainsi $f(a_{k+1}) < f(a_k)$ et l'ordonnée du point $(a_{k+1}, f(a_{k+1}))$ est donc inférieure à celle du point $(a_k, f(a_k))$.
Par contre, si f est décroissante alors $f'(a) < 0$ et

$$a_{k+1} = a_k - \delta f'(a_k) \quad \text{donne} \quad a_{k+1} > a_k,$$

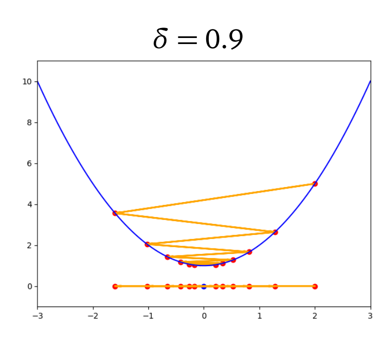
ce qui implique de nouveau $f(a_{k+1}) < f(a_k)$ (car f est décroissante).
Dans tous les cas, l'ordonnée du point $(a_{k+1}, f(a_{k+1}))$ est inférieure à celle du point $(a_k, f(a_k))$.



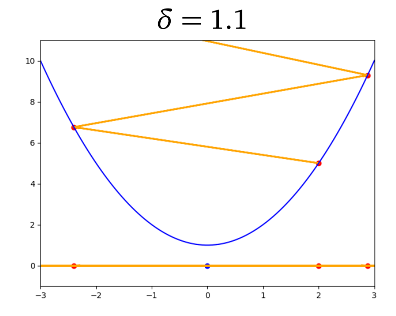


Exemple.
Le choix du paramètre δ est important. Reprenons la fonction f définie par $f(x) = x^2 + 1$ et testons différentes « mauvaises » valeurs du pas δ (avec toujours $a_0 = 2$).

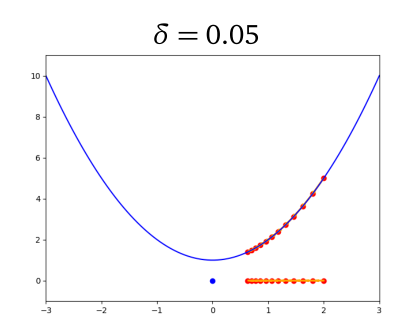
$\delta = 0.9$



$\delta = 1.1$



$\delta = 0.05$



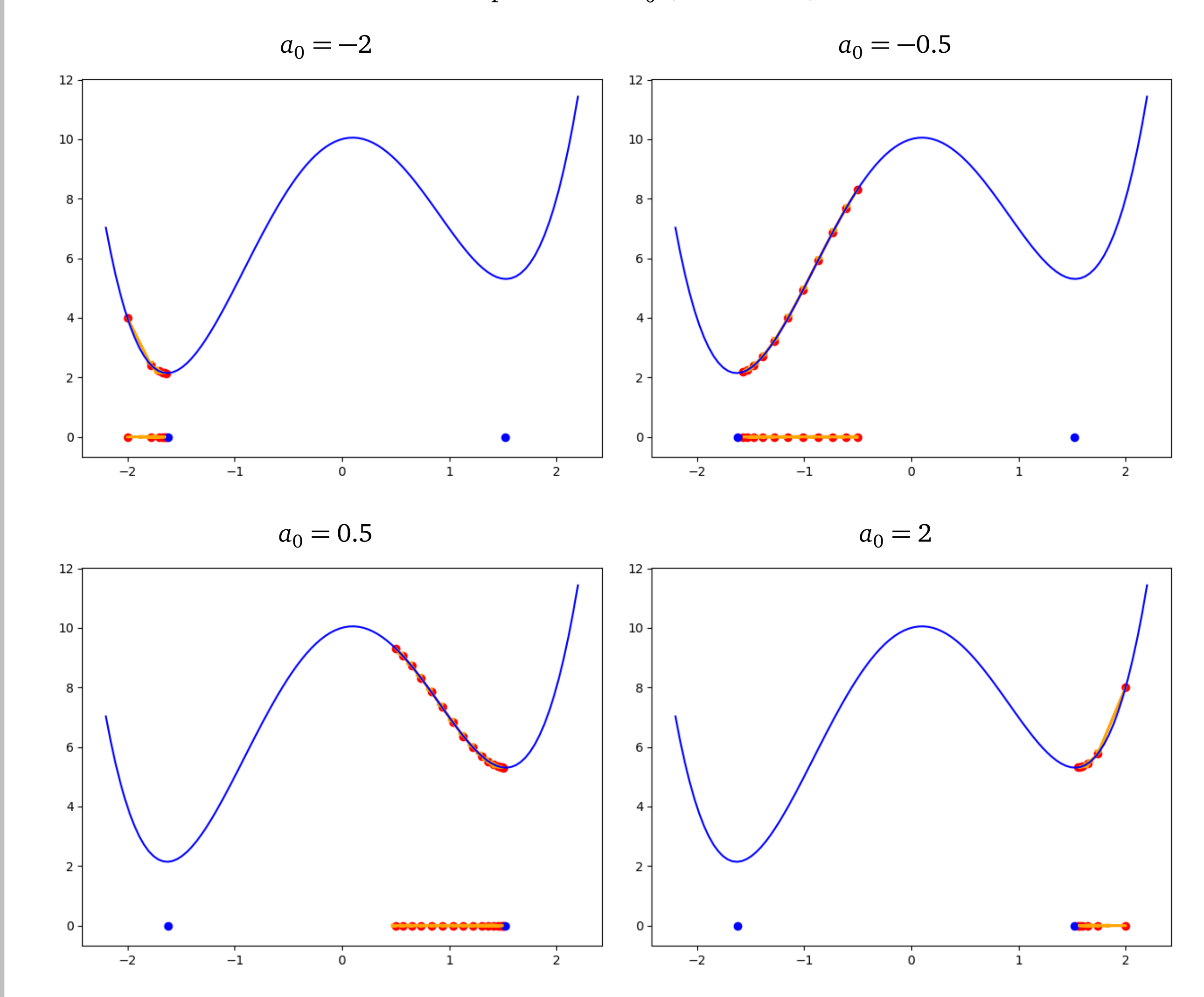
- Pour $\delta = 0.9$, la suite (a_k) tend bien vers $a_{\min} = 0$. Les ordonnées sont bien décroissantes mais comme δ est trop grand, la suite des points oscille de part et d'autre du minimum.
- Pour $\delta = 1.1$, la suite (a_k) diverge. Les ordonnées augmentent, la suite des points oscille et s'échappe. Cette valeur de δ ne donne pas de convergence vers un minimum.
- Pour $\delta = 0.05$, la suite (a_k) tend bien vers $a_{\min}$ mais, comme δ est trop petit, il faudrait beaucoup d'itérations pour arriver à une approximation raisonnable.

Exemple.
Le choix du point de départ est également important surtout lorsqu'il existe plusieurs minimums locaux.

Soit la fonction f définie par :

$$f(a) = a^4 - 5a^2 + a + 10.$$

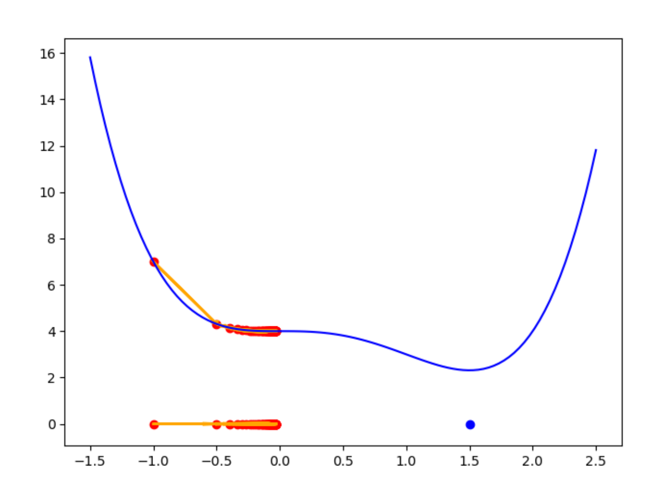
Cette fonction admet deux minimums locaux. La suite (a_k) de la descente de gradient converge vers l'un de ces deux minimums selon le choix du point initial a_0 (ici $\delta = 0.02$).



Exemple.
Les points-selles posent également problème.
La fonction f définie par

$$f(a) = a^4 - 2a^3 + 4$$

a pour dérivée $f'(a) = 4a^3 - 6a^2$ qui s'annule en $a = 0$ qui est l'abscisse d'un point-selle (ni un minimum ni un maximum, en fait la fonction est strictement décroissante autour de $a = 0$). La dérivée s'annule aussi en $a = \frac{3}{2}$ où est atteint le minimum global.
Voici les 100 premières itérations pour la descente de gradient en partant de $a_0 = -1$ (avec $\delta = 0.05$) : la suite a_k converge vers 0 qui n'est pas le minimum recherché.



1.4. Algorithme du gradient

Formalisons un peu les choses pour mettre en évidence l'idée générale et les problèmes techniques qui surviennent.

Algorithme de la descente de gradient.

Soit une fonction $f : \mathbb{R}^n \to \mathbb{R}$, $P \mapsto f(P)$ de plusieurs variables, avec $P = (a_1, \ldots, a_n)$, dont on sait calculer le gradient $\operatorname{grad} f(P)$.

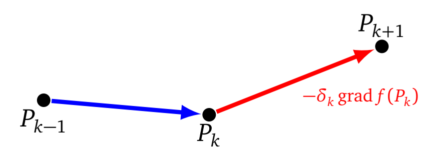
Données.

- Un point initial $P_0 \in \mathbb{R}^n$.
- Un niveau d'erreur $\epsilon > 0$.

Itération. On calcule une suite de points $P_1, P_2, \ldots \in \mathbb{R}^n$ par récurrence de la façon suivante. Supposons que l'on ait déjà obtenu le point P_k :

- on calcule $\operatorname{grad} f(P_k)$,
- on choisit un pas δ_k et on calcule

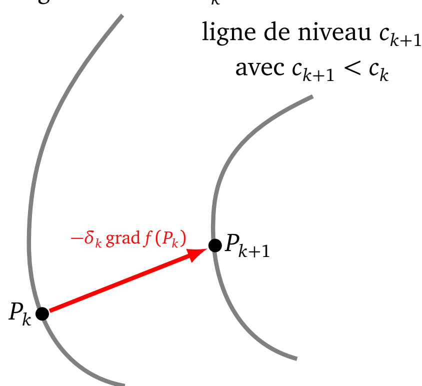
$$P_{k+1} = P_k - \delta_k \operatorname{grad} f(P_k).$$



Arrêt. On s'arrête lorsque $\|\operatorname{grad} f(P_k)\| \leqslant \epsilon$.

Remarque.

- Évidemment, plus on choisit le point initial P_0 proche d'un minimum local, plus l'algorithme va aboutir rapidement. Mais comme on ne sait pas où est ce minimum local (c'est ce que l'on cherche), le plus simple est de choisir un P_0 au hasard.
- Le choix du pas δ_k est crucial. On sait que l'on peut choisir δ_k assez petit de façon à avoir $f(P_{k+1}) \leqslant f(P_k)$ car dans la direction de $-\operatorname{grad} f(P_k)$ la fonction f décroît.



On peut fixer à l'avance un pas δ commun à toutes les itérations, par exemple $\delta = 0.01$. On pourrait également tester à chaque itération plusieurs valeurs de δ par balayage ($\delta = 0.001$, puis $\delta = 0.002\ldots$) et choisir pour δ_k celui en lequel f prend la plus petite valeur.

- Le critère d'arrêt assure qu'en P_k le gradient est très petit. Cela ne garantit pas que ce point soit proche d'un minimum local (et encore moins d'un minimum global). Souvenez-vous : en un minimum local le gradient est nul, mais ce n'est pas parce que le gradient est nul que l'on a atteint un minimum local, cela pourrait être un point-selle voire un maximum local.
  Dans la pratique, on ne définira pas de seuil d'erreur ϵ, mais un nombre d'itérations fixé à l'avance.
- Il est important de calculer $\operatorname{grad} f(a_1, \ldots, a_n)$ rapidement. On pourrait bien sûr calculer une approximation de chacune des dérivées partielles $\frac{\partial f}{\partial a_i}(a_1, \ldots, a_n)$ comme un limite. Mais pour gagner en temps et en précision, on préfère que ce calcul soit fait à l'aide de son expression exacte.

2. Optimisation

Nous allons d'une part résoudre des problèmes d'optimisation : quelle droite approche au mieux un nuage de points, et d'autre part étudier comment améliorer le choix du pas δ.

2.1. Faire varier le pas

On se concentre d'abord sur le choix du ***pas*** δ (*learning rate*).
Rappelons tout d'abord que lorsque l'on se rapproche d'un point minimum, le gradient tend vers 0. Le vecteur $\delta \operatorname{grad} f(P_k)$ tend donc vers 0 à l'approche du minimum, même si δ reste constant.
Cependant, il faut choisir δ ni trop grand, ni trop petit : δ ne doit pas être trop grand car sinon les points P_k vont osciller autour du minimum, mais si δ est trop petit alors les points P_k ne s'approcheront du minimum qu'au bout d'un temps très long. Une solution est de faire varier δ. Pour les premières itérations, on choisit un δ_k assez grand, puis de plus en plus petit au fil des itérations.
Voici différentes formules possibles, à chaque fois δ_0 est le pas initial (par exemple $\delta_0 = 0.1$ ou $\delta_0 = 0.01$).

Décroissance linéaire.

$$\delta_k = \frac{\delta_0}{k+1}.$$

Décroissance quadratique.

$$\delta_k = \frac{\delta_0}{(k+1)^2}.$$

Décroissance exponentielle.

$$\delta_k = \delta_0 e^{-\beta k}$$

où β est une constante positive.

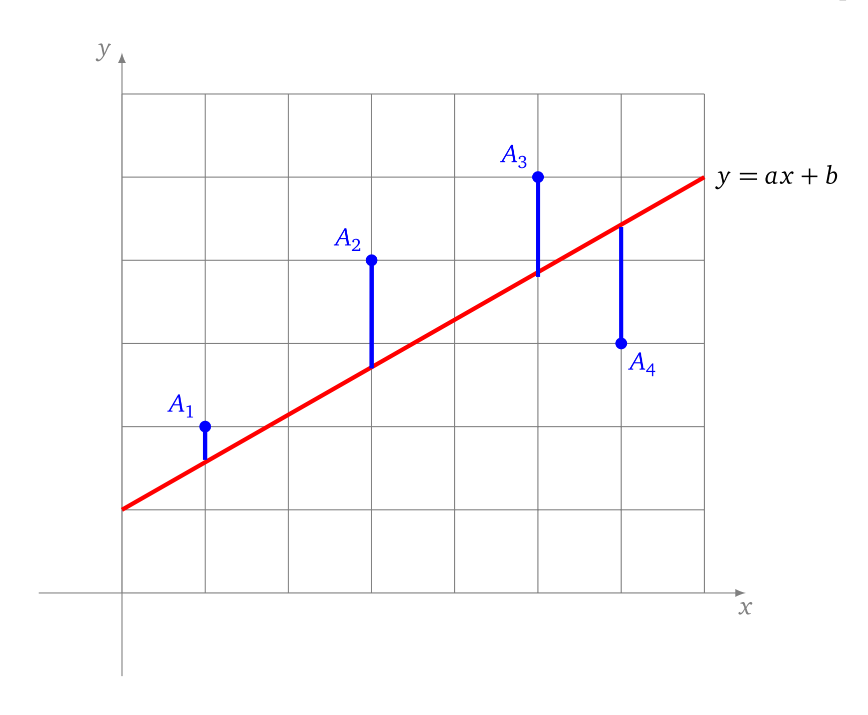
Décroissance linéaire utilisée par *keras*

$$\delta_k = \frac{\delta_0}{\alpha k + 1}$$

où $\alpha \geqslant 0$ est une constante (appelée *decay*). Si $\alpha = 0$ alors δ_k est constant (et vaut δ_0). L'usage courant est d'utiliser des valeurs de α entre 10^{-4} et 10^{-6}.
Terminons par rappeler que le bon choix d'un δ ou des δ_k n'a rien d'évident, il s'obtient soit par test à la main, soit par des expérimentations automatiques, mais à chaque fois il doit être adapté à la situation.

2.2. Régression linéaire $y = ax + b$

On considère un ensemble de N points $A_i = (x_i, y_i)$, $i = 1, \ldots, N$. L'objectif est de trouver l'équation $y = ax + b$ de la droite qui approche au mieux tous ces points. Précisons ce que veut dire « approcher au mieux » : il s'agit de minimiser la somme des carrés des distances verticales entre les points et la droite.



La formule qui donne l'erreur est :

$$E(a,b) = \sum_{i=1}^{N} \big(y_i - (ax_i + b)\big)^2,$$

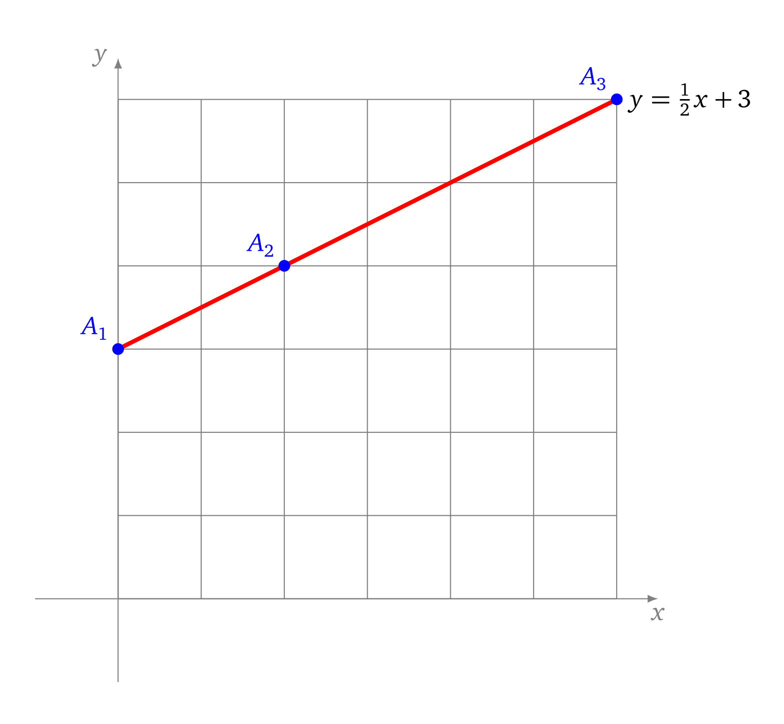
autrement dit

$$E(a,b) = \big(y_1 - (ax_1 + b)\big)^2 + \cdots + \big(y_N - (ax_N + b)\big)^2.$$

Remarquons que l'on a toujours $E(a,b) \geqslant 0$. Si par exemple tous les points sont alignés, alors on peut trouver a et b tels que $E(a,b) = 0$. Quand ce n'est pas le cas, on cherche a et b qui rendent $E(a,b)$ le plus petit possible. Il s'agit donc bien ici de minimiser une fonction de deux variables (les variables sont a et b).
Nous allons appliquer la méthode de la descente de gradient à la fonction $E(a,b)$. Pour cela nous aurons besoin de calculer son gradient :

$$\operatorname{grad} E(a,b) = \left(\frac{\partial E}{\partial a}(a,b), \frac{\partial E}{\partial b}(a,b)\right) = \left(\sum_{i=1}^{N} -2x_i\big(y_i - (ax_i + b)\big), \sum_{i=1}^{N} -2\big(y_i - (ax_i + b)\big)\right).$$

Exemple.
Prenons d'abord l'exemple de trois points $A_1 = (0,3)$, $A_2 = (2,4)$ et $A_3 = (6,6)$ qui sont alignés.



La fonction $E(a, b)$ s'écrit :

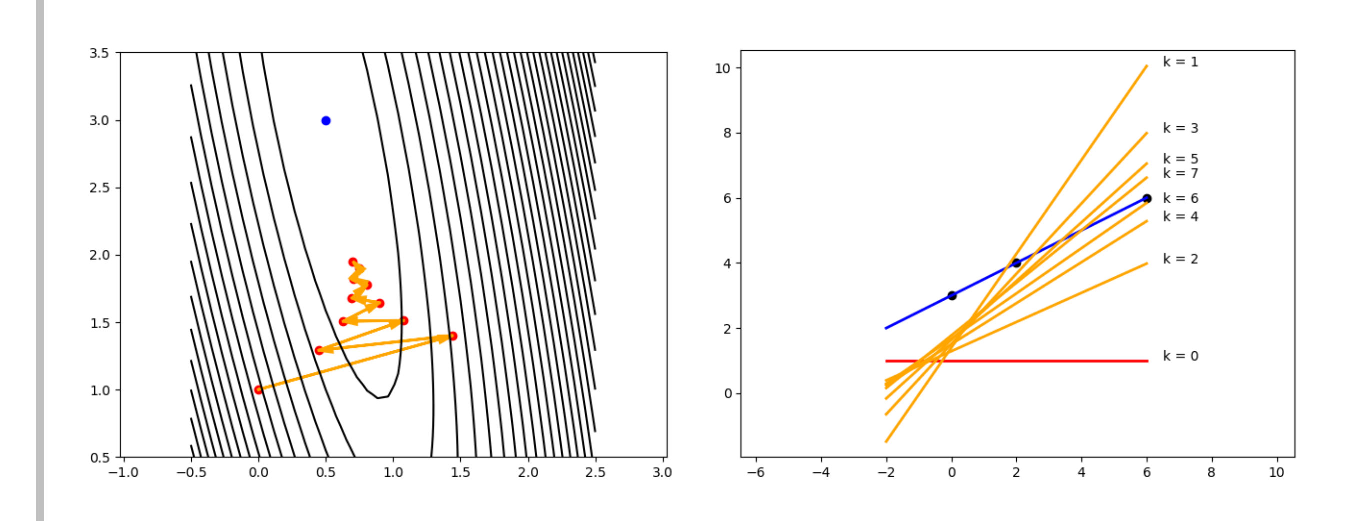
$$E(a, b) = (3 - b)^2 + (4 - (2a + b))^2 + (6 - (6a + b))^2.$$

Partons arbitrairement de $(a_0, b_0) = (0, 1)$ (qui correspond à la droite horizontale d'équation $y = 1$). Voici les valeurs successives (a_k, b_k) obtenues par la méthode de descente de gradient pour un pas $\delta = 0.02$.

k	(a_k, b_k)	$\operatorname{grad} E(a_k, b_k)$	$E(a_k, b_k)$
0	$(0, 1)$	$(-72 - 20)$	38
1	$(1.44, 1.4)$	$(49.60, 5.44)$	18.96
2	$(0.44, 1.29)$	$(-31.50, -11.08)$	10.28
3	$(1.07, 1.51)$	$(22.44, 0.32)$	6.24
4	$(0.62, 1.50)$	$(-13.57, -6.89)$	4.27
5	$(0.90, 1.64)$	$(10.35, -1.72)$	3.24

Au bout de 100 itérations, on obtient $a_{100} \simeq 0.501$ et $b_{100} \simeq 2.99$ (avec un gradient et une erreur presque nuls). C'est bien la droite $y = \frac{1}{2}x + 3$ qui passe par les trois points.

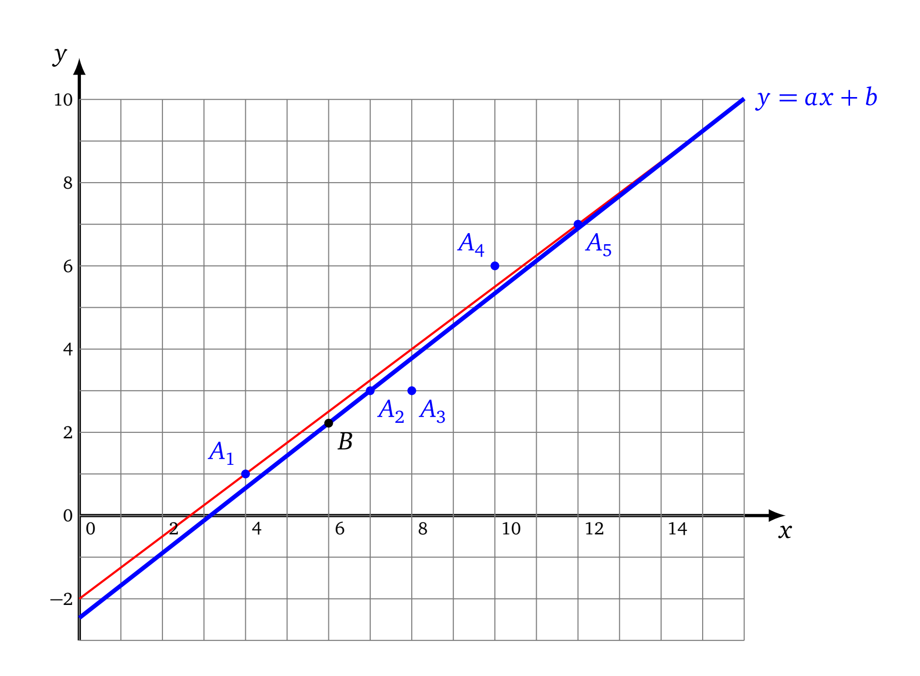
Sur la figure de gauche ci-dessous, sont dessinés, dans le plan de coordonnées (a, b), les premiers points (a_k, b_k) qui convergent (lentement et en oscillant) vers $(\frac{1}{2}, 3)$ (point bleu). Sur la figure de droite sont tracées, dans le plan de coordonnées (x, y), les droites d'équation $y = a_k x + b_k$ pour les premières valeurs de k. Il est beaucoup plus difficile d'appréhender la convergence des droites (vers la droite d'équation $y = \frac{1}{2}x + 3$ en bleu) que celle des points de la figure de gauche.



Exemple.
À partir des données des 5 points suivants, quelle ordonnée peut-on extrapoler pour le point d'abscisse $x = 6$?

$$A_1 = (4,1), \quad A_2 = (7,3), \quad A_3 = (8,3), \quad A_4 = (10,6), \quad A_5 = (12,7).$$

Ces 5 points sont à peu près alignés. On calcule la meilleure droite de régression linéaire par la descente de gradient. Cela revient par exemple à minimiser la fonction $E(a,b)$ présentée ci-dessus, mais actualisée avec nos données. On fixe un pas $\delta = 0.001$. On ne choisit pas le point initial (a_0, b_0) au hasard. Plus on part d'un point proche de la solution, plus la suite convergera rapidement. On trace la droite qui passe par le premier point A_1 et le dernier point A_5. Cette droite a pour équation $y = \frac{3}{4}x - 2$ et est déjà une droite qui approche assez bien les 5 points. Prenons cette droite comme point de départ, c'est-à-dire posons $(a_0, b_0) = (\frac{3}{4}, -2)$. La descente de gradient conduit au bout de 1000 itérations à $a \simeq 0.78$ et $b \simeq -2.46$, pour l'équation de la droite de régression linéaire.
Sur le dessin ci-dessous sont tracées la droite initiale qui passe par les points A_1 et A_5 (en rouge) et la droite de régression linéaire (en bleu).

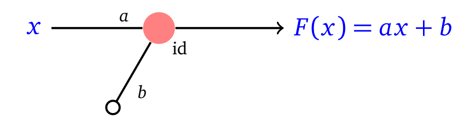


N'oublions pas de répondre à la question initiale. Selon notre modèle linéaire, pour $x = 6$, on doit avoir $y = ax + b \simeq 2.22$ (le point B de la figure ci-dessus).

Remarque : il existe une formule directe pour calculer exactement les coefficients a et b de la droite de régression linéaire, mais ce n'est pas l'esprit de ce cours.

2.3. Régression linéaire et neurone

Quel est le lien entre la régression linéaire et un neurone ? En fait, le neurone suivant a pour fonction associée $F(x) = ax + b$.



Reformulons le problème de la régression linéaire ainsi : soit des données (x_i, y_i), $i = 1, \ldots, N$. Il s'agit de trouver pour le type de neurone choisi les coefficients a et b, tels que la fonction F associée au neurone réalise au mieux ces données, c'est-à-dire

$$F(x_i) \simeq y_i.$$

Vocabulaire :

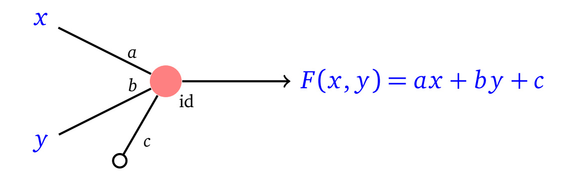
- Chaque x_i est une ***entrée***,
- y_i est la ***sortie attendue*** pour l'entrée x_i,
- $F(x_i)$ est la ***sortie produite***,
- $E_i = (y_i - F(x_i))^2$ est l'erreur pour l'entrée x_i,
- ***L'erreur totale*** est

$$E = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N} (y_i - F(x_i))^2.$$

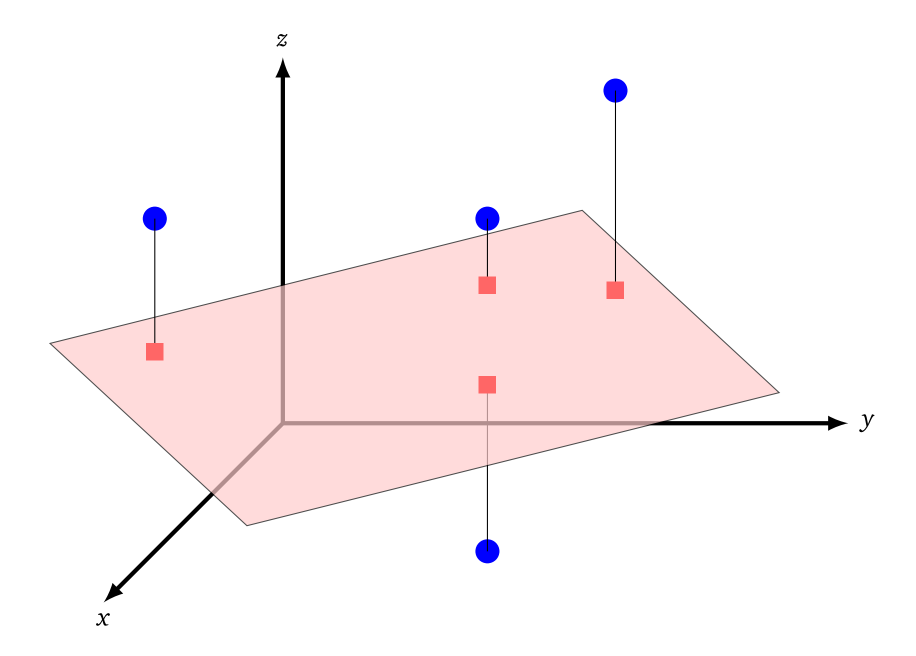
- Vu que $F(x) = ax + b$, on retrouve bien pour E la forme de la fonction présentée précédemment.

2.4. Régression linéaire $z = ax + by + c$

Le neurone suivant a pour fonction $F(x, y) = ax + by + c$. Le problème de la régression linéaire pour un tel neurones et des données (x_i, y_i, z_i), $i = 1, \dots, N$, consiste à trouver des poids (a, b, c) tels que $F(x_i, y_i) \simeq z_i$.



En terme géométrique, il s'agit de trouver un plan d'équation $z = ax + by + c$ qui approche au mieux des points donnés de l'espace (x_i, y_i, z_i).



La méthode est la même que précédemment, il s'agit de minimiser la fonction erreur de trois variables :

$$E(a, b, c) = \sum_{i=1}^{N} \left(z_i - (ax_i + by_i + c)\right)^2.$$

On aura besoin de calculer le gradient de E :

$$\operatorname{grad} E(a, b, c) = \left(\frac{\partial E}{\partial a}(a, b, c), \frac{\partial E}{\partial b}(a, b, c), \frac{\partial E}{\partial c}(a, b, c)\right).$$

On a :

$$\operatorname{grad} E(a, b, c) = \left(\sum_{i=1}^{N} -2x_i\big(z_i - (ax_i + by_i + c)\big), \sum_{i=1}^{N} -2y_i\big(z_i - (ax_i + by_i + c)\big), \sum_{i=1}^{N} -2\big(z_i - (ax_i + by_i + c)\big)\right).$$

Exemple.
Voici un exemple avec trois points $A_1 = (0, 0, 3)$, $A_2 = (1, 0, 4)$ et $A_3 = (0, 1, 5)$. Trois points appartiennent nécessairement à un plan. Nous cherchons donc l'équation $z = ax + by + c$ de ce plan. La fonction d'erreur est :

$$E(a, b, c) = (3 - c)^2 + \big(4 - (4a + c)\big)^2 + \big(5 - (b + c)\big)^2.$$

La descente de gradient appliquée à cette fonction avec un pas $\delta = 0.2$ et des valeurs initiales $(a_0, b_0, c_0) = (0, 0, 0)$ conduit à une suite (a_k, b_k, c_k) qui tend vers $(a, b, c) = (1, 2, 3)$. Le plan recherché est donc $z = x + 2y + 3$.

Exemple.
Voici un jeu de cinq données
$$(1,0,0), \quad (0,1,5), \quad (2,1,1), \quad (1,2,0), \quad (2,2,3).$$
À partir des données des 5 points ci-dessus, quelle est la troisième coordonnée z du point tel que $x = 1$ et $y = 1$ du plan qui les extrapole ?
Les points ne sont pas coplanaires. On cherche donc le plan qui approche au mieux ces cinq points. La descente de gradient pour la fonction $E(a, b, c)$ avec $\delta = 0.01$ et un point initial $(a_0, b_0, c_0) = (0,0,0)$ conduit au plan $z = ax + by + c$ avec $a \simeq -1.22$, $b \simeq 0.77$, $c \simeq 2.33$ pour lesquels la valeur de E est minimale avec $E(a, b, c) \simeq 14.44$.
Pour $(x, y) = (1,1)$, la sortie produite par le modèle linéaire est $z = ax + by + c \simeq 1.88$.

3. Descente de gradient stochastique

La descente de gradient stochastique (abrégée en *sgd*) est une façon d'optimiser les calculs de la descente de gradient pour une fonction d'erreur associée à une grande série de données. Au lieu de calculer un gradient (compliqué) et un nouveau point pour l'ensemble des données, on calcule un gradient (simple) et un nouveau point par donnée, il faut répéter ce processus pour chaque donnée.

3.1. Petits pas à petits pas

Revenons à l'objectif visé par la régression linéaire.
On considère des données (X_i, y_i), $i = 1, \ldots, N$ où $X_i \in \mathbb{R}^\ell$ et $y_i \in \mathbb{R}$. Ces données proviennent d'observations ou d'expérimentations.
Il s'agit de trouver une fonction $F : \mathbb{R}^\ell \to \mathbb{R}$ qui modélise au mieux ces données, c'est-à-dire telle que
$$F(X_i) \simeq y_i.$$
Pour ***l'entrée*** X_i, la valeur y_i est la ***sortie attendue***, alors que $F(X_i)$ est la ***sortie produite*** par notre modèle.
Pour mesurer la pertinence de la fonction F, on introduit la fonction d'***erreur totale*** qui mesure l'écart entre la sortie attendue et la sortie produite :
$$E = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N} (y_i - F(X_i))^2.$$
Cette erreur totale est une somme d'***erreurs locales*** :
$$E_i = (y_i - F(X_i))^2.$$
Le but du problème est de déterminer la fonction F qui minimise l'erreur E. Par exemple, dans le cas de la régression linéaire, il fallait trouver les paramètres a et b pour définir $F(x) = ax + b$, ou bien, pour deux variables, les paramètres a, b, c pour définir $F(x, y) = ax + by + c$.
Considérons une fonction erreur $E : \mathbb{R}^n \to \mathbb{R}$ qui dépend de n paramètres $a_1, \ldots, a_n$ (qui définissent l'expression de la fonction F).

Descente de gradient classique. Pour minimiser l'erreur et déterminer les meilleurs paramètres, on peut appliquer la méthode du gradient classique.
On part d'un point $P_0 = (a_1, \ldots, a_n) \in \mathbb{R}^n$, puis on applique la formule de récurrence :
$$P_{k+1} = P_k - \delta \operatorname{grad} E(P_k).$$

Pour appliquer cette formule, il faut calculer des gradients $\operatorname{grad} E(P_k)$, or

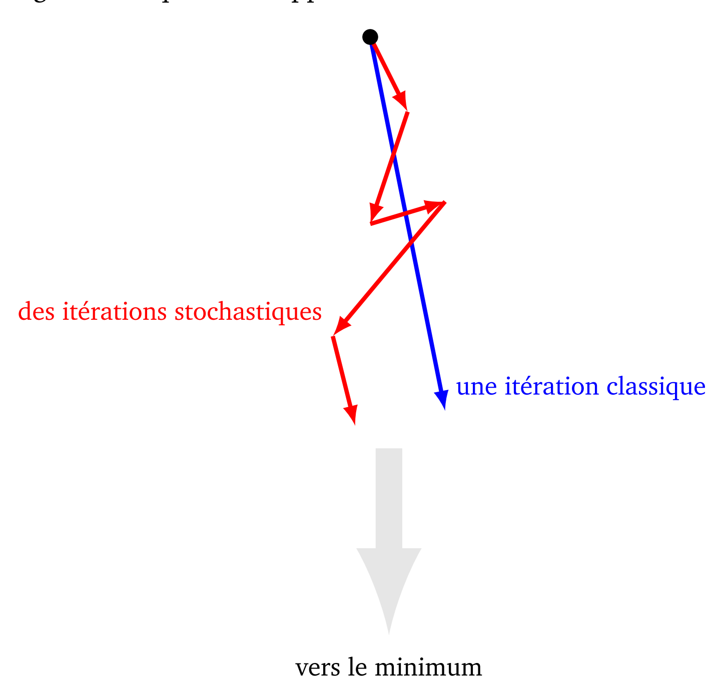
$$\operatorname{grad} E(P_k) = \sum_{i=1}^{N} \operatorname{grad} E_i(P_k).$$

Il faut donc calculer une somme de N termes à chaque itération, ce qui pose des problèmes d'efficacité pour de grandes valeurs de N.

Descente de gradient stochastique.
Pour diminuer la quantité de calculs, l'idée est de considérer à chaque itération un seul gradient E_i à la place de E. C'est-à-dire :

$$P_{k+1} = P_k - \delta \operatorname{grad} E_i(P_k)$$

pour *une seule erreur* E_i (correspondant à la donnée numéro i). L'itération suivante se basera sur l'erreur E_{i+1}.
Quel est l'intérêt de cette méthode ? Dans la méthode de gradient classique, on calcule à chaque itération un « gros » gradient (associé à la totalité des N données) qui nous rapproche d'un grand pas vers le minimum. Ici on calcule N « petits » gradients qui nous rapprochent du minimum.



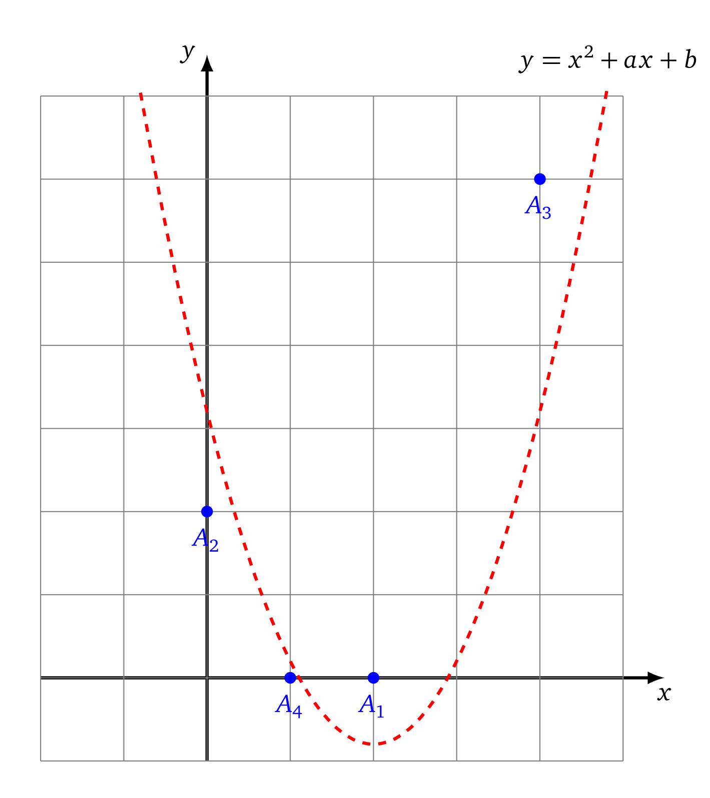
Voici les premières itérations de cet algorithme.

- On part d'un point P_0.
- On calcule $P_1 = P_0 - \delta \operatorname{grad} E_1(P_0)$. C'est la formule du gradient, mais seulement pour l'erreur locale E_1 (juste à partir de la première donnée (X_1, y_1)).
- On calcule $P_2 = P_1 - \delta \operatorname{grad} E_2(P_1)$. C'est la formule du gradient, mais seulement pour l'erreur locale E_2.
- On itère encore et encore.
- On calcule $P_N = P_{N-1} - \delta \operatorname{grad} E_N(P_{N-1})$. C'est la formule du gradient, mais seulement pour l'erreur locale E_N. À ce stade de l'algorithme, nous avons tenu compte de toutes les données.
- On calcule $P_{N+1} = P_N - \delta \operatorname{grad} E_1(P_N)$. On recommence pour P_N et l'erreur locale E_1.
- Etc. On s'arrête au bout d'un nombre d'étapes fixé à l'avance ou lorsque l'on est suffisamment proche du minimum.

Exemple.
On considère les quatre points :

$$A_1 = (2,0), \quad A_2 = (0,2), \quad A_3 = (4,6), \quad A_4 = (1,0).$$

Comme ces points ne sont clairement pas alignés, on cherche un modèle pour les placer au mieux sur une parabole.



On va ici chercher des coefficients a et b tels que les points soient proches de la parabole d'équation $y = x^2 + ax + b$. On note donc $F(x) = x^2 + ax + b$ et on souhaite $F(x_i) \simeq y_i$ pour les points $A_i = (x_i, y_i)$, $i = 1, \dots, 4$.
Les fonctions d'erreurs locales sont :

$$E_i(a, b) = \big(y_i - (x_i^2 + ax_i + b)\big)^2.$$

La fonction d'erreur globale est :

$$E(a, b) = E_1(a, b) + E_2(a, b) + E_3(a, b) + E_4(a, b).$$

Voici les premières itérations pour chacune des méthodes, la descente de gradient classique (à gauche), la descente de gradient stochastique (à droite) toutes les deux en partant du point $(a_0, b_0) = (1, 1)$ et avec $\delta = 0.01$.

Descente de gradient

k	(a_k, b_k)	$E(a_k, b_k)$
0	$(1, 1)$	284
1	$(-0.54, 0.52)$	84.87
2	$(-1.36, 0.29)$	28.68

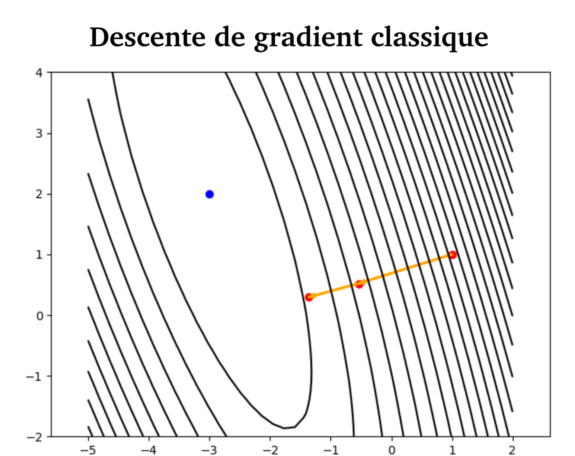
Descente de gradient stochastique

k	(a'_k, b'_k)	$E(a'_k, b'_k)$
0	$(1, 1)$	284
1	$(0.72, 0.86)$	236.43
2	$(0.72, 0.88)$	237.41
3	$(-0.38, 0.60)$	100.74
4	$(-0.40, 0.58)$	97.917
5	$(-0.55, 0.50)$	83.245
6	$(-0.55, 0.53)$	83.913
7	$(-1.22, 0.37)$	36.48
8	$(-1.22, 0.36)$	36.29

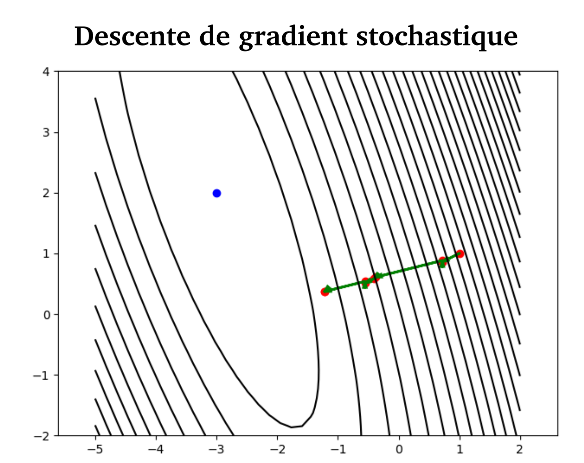
Au bout de 200 itérations la descente de gradient classique conduit à $(a_{200}, b_{200}) \simeq (-2.9981, 1.9948)$ (chaque donnée a été utilisée 200 fois).

Cela correspond à 800 itérations de la descente de gradient stochastique (chacune des 4 données a été utilisée 200 fois) cela conduit à $(a'_{800}, b'_{800}) \simeq (-2.9984, 1.9954)$. La limite cherchée étant $(a, b) = (-3, 2)$, avec $E(a, b) = 0$, les deux méthodes convergent à la même vitesse. Chaque calcul de gradient de la méthode stochastique est très simple, mais il faut plus d'itérations.
Voici les points des premières itérations correspondant au tableau ci-dessus.

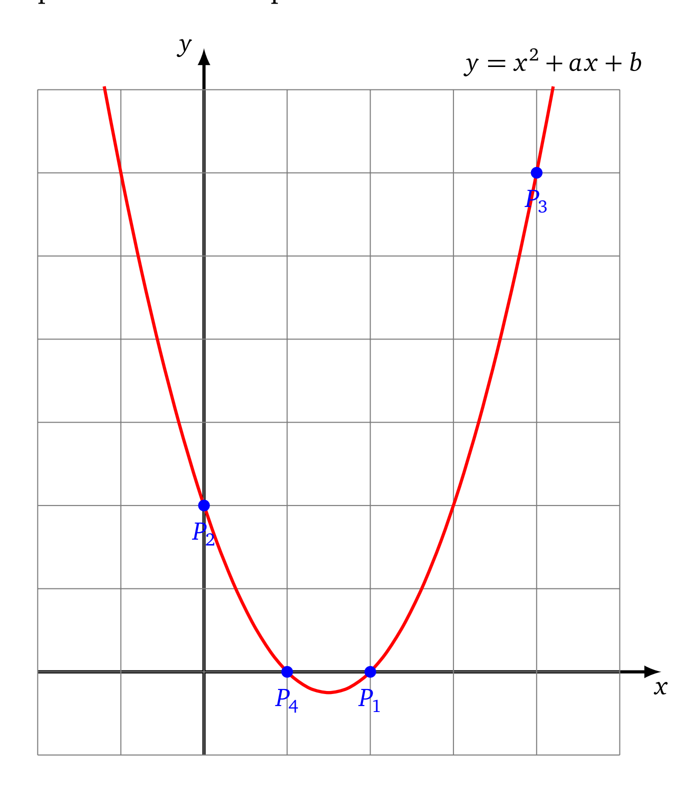
Descente de gradient classique



Descente de gradient stochastique



Conclusion : sur cet exemple les points sont exactement sur la parabole d'équation $y = x^2 + ax + b$ avec $a = -3$ et $b = 2$. Bien sûr cette méthode est encore plus intéressante lorsqu'il s'agit de trouver une parabole qui ne contient pas l'ensemble des points donnés.



Terminons par des remarques plus techniques. Tout d'abord, la formule précise de la descente de gradient stochastique est :

$$P_{k+1} = P_k - \delta \operatorname{grad} E_{(k\%N)+1}(P_k)$$

où $k\%N$ est « k modulo N ».
La descente de gradient stochastique est une méthode qui peut être plus efficace :

- Tout d'abord elle n'utilise qu'une donnée à la fois et évite ainsi les problèmes de mémoire de la descente classique pour laquelle il faut manipuler toutes les données à chaque itération.
- Toujours dans le cas où l'on a beaucoup de données, la descente de gradient stochastique peut converger

en deux ou trois passages sur l'ensemble des données, alors que la descente classique nécessite toujours plusieurs itérations (voir la section 3.3 plus loin).

- Avec la méthode stochastique, on calcule des gradients en des points qui sont plus proches du minimum. Attention cependant, certains petits pas peuvent aller dans la mauvaise direction.
- Le caractère aléatoire de ces petits pas est parfois un avantage, par exemple pour s'échapper d'un point-selle.

3.2. Différentes fonctions d'erreurs

Il existe différentes formules pour calculer l'erreur entre la sortie attendue y_i et la sortie produite $F(x_i)$. On considère une série de valeurs y_i, $i = 1, \ldots, N$ (fournie par observations ou expérimentations) qui sont approchées par des valeurs $F(x_i)$ produites par une formule issue d'un réseau de neurones par exemple. Le but est d'obtenir $F(x_i)$ le plus proche possible de y_i, pour tout $i = 1, \ldots, N$. Pour savoir si l'objectif est atteint, on mesure l'écart entre ces valeurs.

Erreur quadratique moyenne.

$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - F(x_i))^2.$$

C'est la formule la plus classique (en anglais *minimal squared error* ou *mse*). Bien entendu $E \geqslant 0$ quels que soit les $F(x_i)$ et $E = 0$ si et seulement si $y_i = F(x_i)$ pour tous les $i = 1, \ldots, N$. C'est presque la formule que l'on a utilisée pour la régression linéaire (il n'y avait pas le facteur $\frac{1}{N}$).

Erreur absolue moyenne.

$$E = \frac{1}{N}\sum_{i=1}^{N}|y_i - F(x_i)|.$$

C'est une formule plus naturelle, mais moins agréable à manipuler à cause de la valeur absolue.

Noter que pour ces deux formules, l'erreur globale E est la moyenne d'erreurs locales $E_i = (y_i - F(x_i))^2$ (ou bien $E_i = |y_i - F(x_i)|$). Les erreurs locales sont indépendantes les unes des autres, ce qui est la base de la descente de gradient stochastique. (Une formule d'erreur du type $E = y_1 y_2 - F(x_1)F(x_2)$ ne permettrait pas la descente stochastique.)

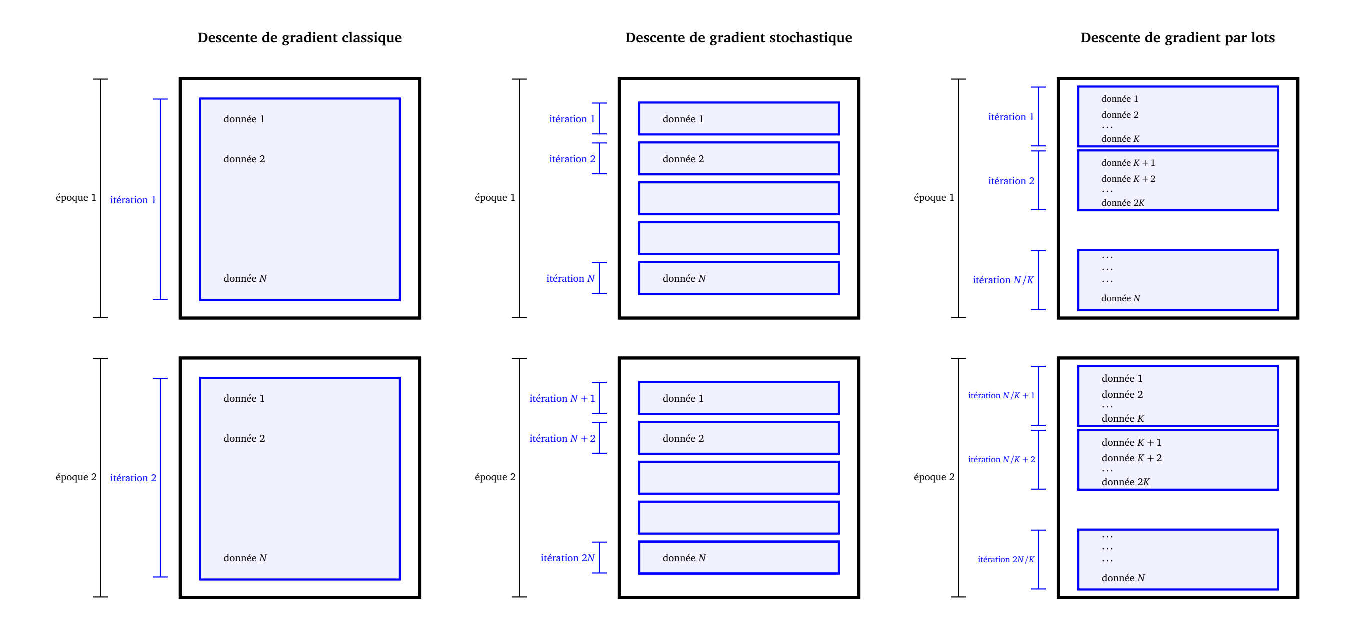
Il existe d'autres formules d'erreur, en particulier si la sortie attendue est du type 0 ou 1 ou bien si la sortie produite est une probabilité $0 \leqslant p \leqslant 1$.

3.3. Descente par lots

Il existe une méthode intermédiaire entre la descente de gradient classique (qui tient compte de toutes les données à chaque itération) et la descente de gradient stochastique (qui n'utilise qu'une seule donnée à chaque itération).

La descente de gradient par ***lots*** (ou ***mini-lots***, *mini-batch*) est une méthode intermédiaire : on divise les données par paquets de taille K. Pour chaque paquet (appelé « lot »), on calcule un gradient et on effectue une itération.

Au bout de N/K itérations, on a parcouru tout le jeu de données : cela s'appelle une ***époque***.



La formule est donc

$$P_{k+1} = P_k - \delta \operatorname{grad}(E_{j_0+1} + E_{j_0+2} + \cdots + E_{j_0+K})(P_k).$$

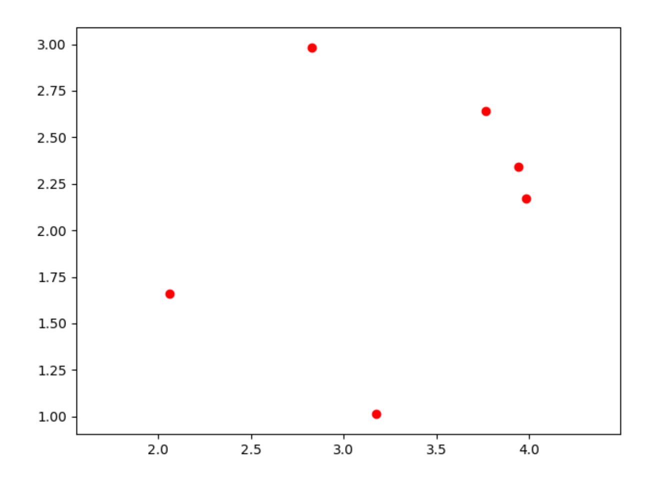
Pour P_{k+2}, on repart de P_{k+1} et on utilise le gradient de la fonction $E_{j_0+K+1} + E_{j_0+K+2} + \cdots + E_{j_0+2K}$.

Remarque.

- Pour $K = 1$, c'est exactement la descente de gradient stochastique. Pour $K = N$, c'est la descente de gradient classique.
- Cette méthode combine le meilleur des deux mondes : la taille des données utilisées à chaque itération peut être adaptée à la mémoire et le fait de travailler par lots évite les pas erratiques de la descente stochastique pure.
- On peut par exemple choisir $2 \leqslant K \leqslant 32$ et profiter du calcul parallèle en calculant $\operatorname{grad}(E_1 + \cdots + E_K)$, par le calcul de chacun des $\operatorname{grad} E_i$ sur K processeurs, puis en additionnant les résultats.
- Il est d'usage de mélanger au hasard les données (X_i, y_i) avant chaque époque.

Exemple.

Voyons un exemple d'interpolation circulaire. Les 6 points ci-dessous sont sur un cercle. Comment déterminer son centre et son rayon ?



Pour des points (x_i, y_i), $i = 1, \ldots, N$, on mesure la distance globale par rapport au cercle $\mathcal{C}$ de centre

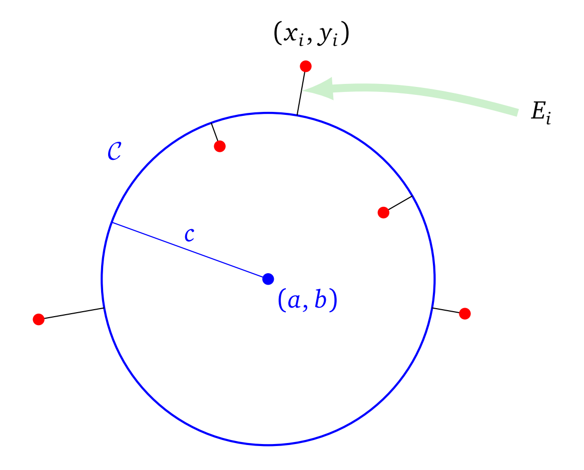
(a, b) et de rayon c par la formule d'erreur :

$$E(a, b, c) = \sum_{i=1}^{N} E_i(a, b, c)$$

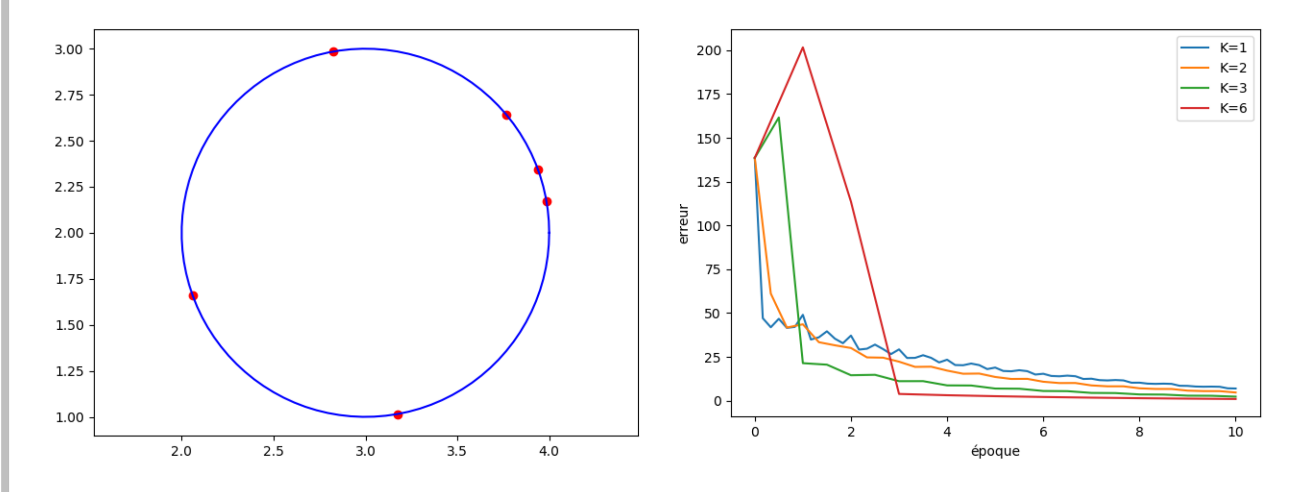
où

$$E_i(a, b, c) = \left((x_i - a)^2 + (y_i - b)^2 - c^2\right)^2.$$

En effet, E_i mesure en quelque sorte la distance entre le point (x_i, y_i) et le cercle $\mathcal{C}$ de centre (a, b) et de rayon c. Donc $E_i = 0$ si et seulement si $(x_i, y_i) \in \mathcal{C}$, sinon $E_i > 0$.

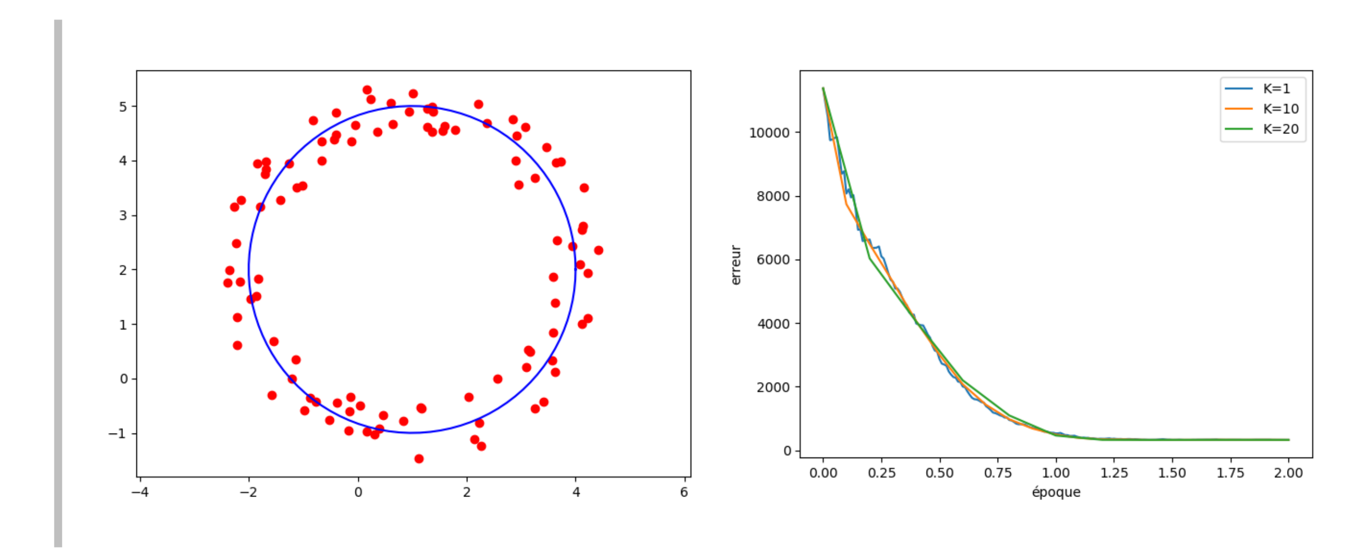


Dans notre exemple, les $N = 6$ points sont exactement situés sur le cercle de centre (a, b) et de rayon c. En appliquant la descente de gradient par lots avec $\delta = 0.01$ et $(a_0, b_0, c_0) = (1, 1, 2)$, on trouve $(a, b) = (3, 2)$ et $c = 1$. Ci-dessous, à droite, nous avons représenté les valeurs de l'erreur totale E (qui tend vers 0) en fonction du nombre d'époques et ceci pour différentes tailles du lot : $K = 1$ (descente stochastique), $K = 2$, $K = 3$ et $K = 6$ (descente classique).



On remarque qu'au bout de 10 époques la valeur de l'erreur est à peu près la même quelle que soit la taille K de l'échantillon. Par contre, l'évolution au départ est différente. Par exemple pour $K = 1$, l'erreur fluctue à la hausse ou à la baisse à chaque itération.

Cette méthode présente bien sûr davantage d'intérêt quand les points ne sont pas exactement sur un cercle. Il s'agit alors de trouver le meilleur cercle qui convient, c'est-à-dire de trouver le minimum (cette fois non nul) de E. Voici un exemple de $N = 100$ points tirés au hasard autour du cercle de centre $(a, b) = (1, 2)$ et de rayon $c = 3$. La descente de gradient est appliquée avec $\delta = 0.001$ et $(a_0, b_0, c_0) = (1, 1, 1)$ pour des lots de différentes tailles $K = 1$, $K = 10$ et $K = 20$. On remarque que deux époques suffisent pour avoir convergence et que plus la taille K de l'échantillon est grande plus la convergence est régulière vers le minimum.

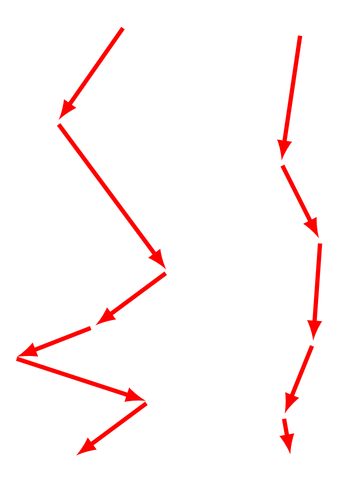


4. Accélérations

Le choix du pas δ n'est pas la seule amélioration possible de la méthode du gradient, nous allons voir comment la modifier à l'aide du « moment ». Commençons par revenir à l'analogie de la descente du gradient classique qui correspond à une goutte d'eau qui descend une montagne : la goutte emprunte le chemin qui suit la courbe de plus grande pente, quitte à serpenter et osciller lors de la descente. Imaginons que l'on lance maintenant une balle assez lourde du haut de la même montagne. Cette balle va suivre, comme la goutte d'eau, le chemin de la plus forte pente, mais une fois lancée elle va acquérir de l'inertie, appelée *moment*, qui va atténuer ses changements de direction. Ainsi la balle ne s'embarrasse pas des petits aléas du terrain et dévale la pente plus rapidement que la goutte d'eau.

Nos petits aléas de terrain à nous viennent du fait que l'on ne calcule pas exactement le gradient de la fonction d'erreur en utilisant tout le jeu de données à chaque fois, mais seulement un échantillon. Cela peut conduire à certains gradients mal orientés. L'inertie de la balle est en quelque sorte la mémoire de la trajectoire passée qui corrige les mouvements erratiques.

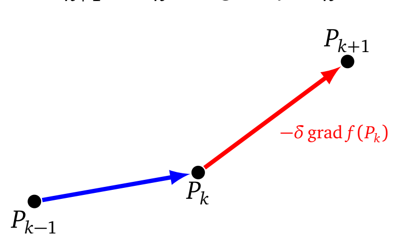
Sur la figure de gauche, la descente classique (la goutte d'eau), sur la figure de droite, la descente de gradient avec le moment (la balle).



4.1. Moment

Rappelons la formule de la descente de gradient classique :

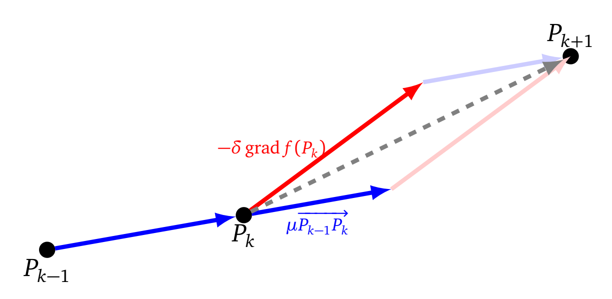
$$P_{k+1} = P_k - \delta \operatorname{grad} f(P_k).$$



Considérons nos points comme une particule qui voyage au cours du temps. Alors le vecteur $\overrightarrow{P_{k-1}P_k}$ correspond à la vitesse de cette particule et est appelé le ***moment*** au point P_k.
La formule de la descente de gradient avec moment est :

$$P_{k+1} = P_k + \mu\overrightarrow{P_{k-1}P_k} - \delta \operatorname{grad} f(P_k).$$

où $\mu, \delta \in \mathbb{R}$. Cette formule peut être définie pour $k = 0$ si on suppose que la particule est immobile au départ, c'est-à-dire en posant $\overrightarrow{P_{-1}P_0} = \vec{0}$.
On peut prendre par exemple $\mu \in [0.5, 0.9]$ et $\delta = 0.01$.
Schématiquement, au point P_k nous avons deux vecteurs : un qui provient du moment $\mu\overrightarrow{P_{k-1}P_k}$ (la mémoire du passé) et un qui provient du gradient $-\delta \operatorname{grad} f(P_k)$ (qui projette vers l'avenir). La somme permet de calculer le point suivant P_{k+1}.



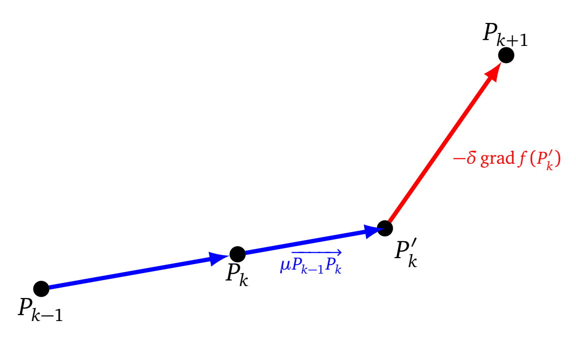
4.2. Nesterov

Dans la méthode précédente, le moment et le gradient sont calculés au même point P_k. La méthode de Nesterov est une variante de cette méthode. Elle consiste à appliquer d'abord le moment, pour obtenir un point P_k', puis de calculer le gradient en ce point (et non en P_k).
La formule est donc

$$P_{k+1} = P_k + \mu\overrightarrow{P_{k-1}P_k} - \delta \operatorname{grad} f(P_k + \mu\overrightarrow{P_{k-1}P_k}).$$

Autrement dit, si on note P_k' le point $P_k + \mu\overrightarrow{P_{k-1}P_k}$ alors

$$P_{k+1} = P_k' - \delta \operatorname{grad} f(P_k').$$



C'est un petit avantage par rapport à la méthode du moment puisqu'on calcule le gradient au point P_k' qui est censé être plus près de la solution $P_{\min}$ que P_k.

4.3. Vocabulaire

Terminons par un petit résumé du vocabulaire avec sa traduction en anglais :

- descente de gradient (classique), *(batch) gradient descent*,
- descente de gradient stochastique, *sgd* pour *stochastic gradient descent*,
- descente de gradient par lots, *mini-batch gradient descent*,
- pas δ, *learning rate*,
- erreur quadratique moyenne, *mse* pour *minimal squared error*,
- moment, *momentum*,
- époque, *epoch*.

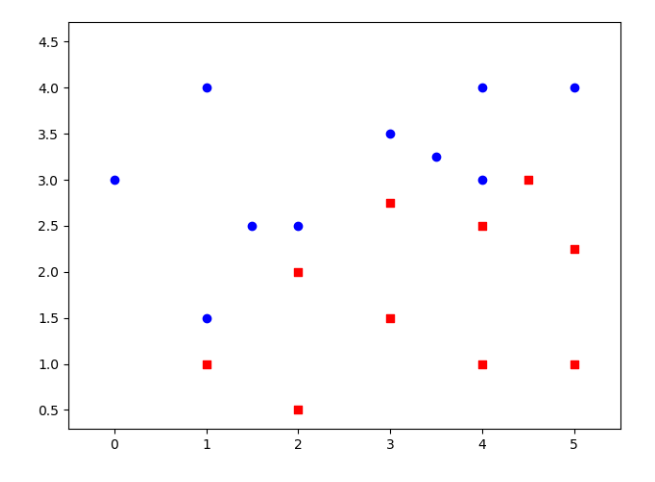
Chapitre 9 Rétropropagation

Vidéo ■ partie 9.1. Principe de la rétropropagation
Vidéo ■ partie 9.2. Exemples de rétropropagation
Vidéo ■ partie 9.3. Sur-apprentissage et autres soucis

La rétropropagation, c'est la descente de gradient appliquée aux réseaux de neurones. Nous allons étudier des problèmes variés et analyser les solutions produites par des réseaux de neurones.

1. Principe de la rétropropagation

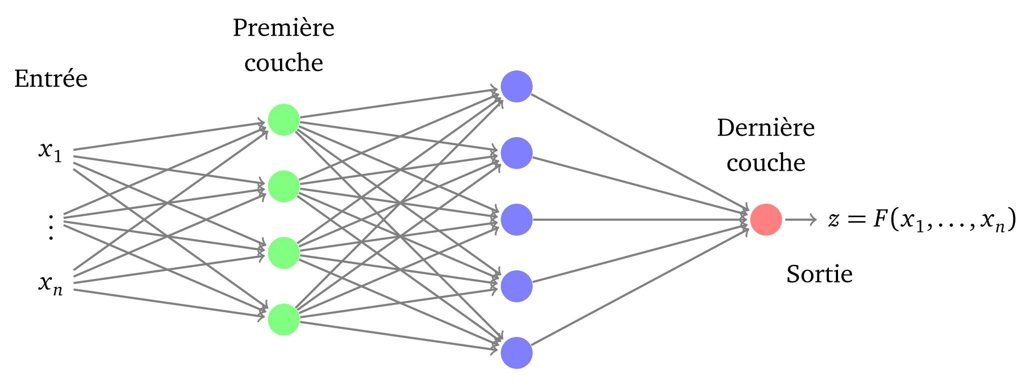
Voici un jeu de données : des ronds bleus et des carrés rouges. Nous souhaitons trouver un modèle simple qui caractérise ces données.



Plus exactement, nous souhaiterions pouvoir dire pour chaque point du plan s'il devrait être colorié en rouge ou bien en bleu et ceci pour des points qui ne sont pas dans les données de départ. De plus, nous voulons ne rien faire à la main, mais que la réponse soit calculée par la machine !
Nous allons revoir pas à pas l'utilisation d'un réseau de neurones pour résoudre un problème et traiterons en particulier l'exemple ci-dessus.

1.1. Objectif du réseau

- Soit $\mathcal{R}$ un réseau de neurones. Celui-ci est défini par son architecture (le nombre de couches, le nombre de neurones par couche), les fonctions d'activation et l'ensemble $P = (a_1, a_2, \ldots)$ des poids de tous les neurones.
- À ce réseau $\mathcal{R}$ on associe une fonction $F : \mathbb{R}^n \to \mathbb{R}^p$ où n est la dimension des entrées (de la première couche) et p le nombre de neurones de la couche de sortie. Dans ce chapitre, nous supposerons qu'il n'y a qu'une seule sortie, c'est-à-dire $p = 1$ et $F : \mathbb{R}^n \to \mathbb{R}$.



- On dispose de données (X_i, z_i) (pour $i = 1, \ldots, N$) où $X_i \in \mathbb{R}^n$ est une ***entrée*** (de la forme $X = (x_1, \ldots, x_n)$) et $z_i \in \mathbb{R}$ est la ***sortie attendue*** pour cette entrée.
- Le but est de trouver les poids du réseau afin que la fonction F qui lui est associée vérifie :

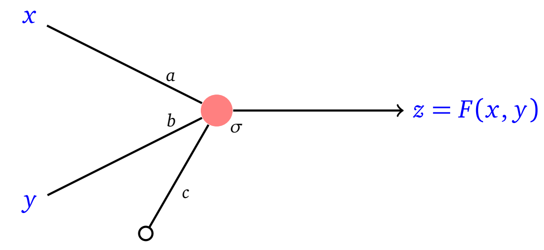
$$F(X_i) \simeq z_i \quad \text{pour tout} \quad i = 1, \ldots, N.$$

- Pour mesurer précisément la performance de l'approximation, on définit une ***fonction erreur*** :

$$E = \frac{1}{N} \sum_{i=1}^{N} E_i \qquad \text{avec} \qquad E_i = \big(F(X_i) - z_i\big)^2.$$

Exemple.
Nous traitons l'exemple donné en introduction.

- On décide de construire le réseau le plus simple possible : avec un seul neurone. Cela correspond à séparer les points du plan selon une droite. On choisit la fonction d'activation σ. La dimension de l'entrée est 2 et celle la sortie est 1. Il y a 3 poids (a, b, c) à calculer pour terminer le paramétrage du réseau.



- Pour chaque triplet de poids (a, b, c) notre réseau définit une fonction $F : \mathbb{R}^2 \to \mathbb{R}$ qui est en fait ici :

$$F(x, y) = \sigma(ax + by + c)$$

avec $\sigma(t) = \frac{1}{1+e^{-t}}$.

- Nos données sont les points rouges et bleus. Une entrée X_i est donc constituée des coordonnées (x_i, y_i) d'un point et la sortie attendue pour ce point est $z_i = 0$ (pour les points bleus) et $z_i = 1$ (pour les points rouges). Voici les coordonnées des ronds bleus (sortie attendue $z_i = 0$) :

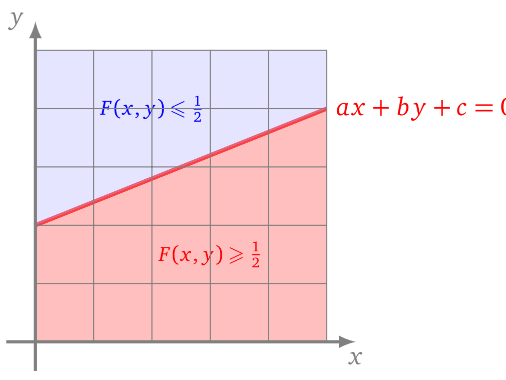
$$(0,3),\ (1,1.5),\ (1,4),\ (1.5,2.5),\ (2,2.5),\ (3,3.5),\ (3.5,3.25),\ (4,3),\ (4,4),\ (5,4)$$

et des carrés rouges (sortie attendue $z_i = 1$) :

$$(1,1),\ (2,0.5),\ (2,2),\ (3,1.5),\ (3,2.75),\ (4,1),\ (4,2.5),\ (4.5,3),\ (5,1),\ (5,2.25).$$

- Comment définir les poids (a, b, c) afin que $F(x_i, y_i) \simeq 0$ pour tous les ronds bleus et que $F(x_i, y_i) \simeq 1$ pour tous les carrés rouges ? Si on sait trouver de tels poids alors on pourra colorier (presque) n'importe quel point (x, y) du plan (et pas seulement nos ronds et nos carrés) : si $F(x, y) \simeq 0$, on coloriera le point en bleu, si par contre $F(x, y) \simeq 1$, on le coloriera en rouge. Plus précisément, la fonction F (qui dépend de σ) a ses valeurs dans $[0, 1]$ et prend la valeur $F(x, y) = \frac{1}{2}$ exactement sur la droite $ax + by + c = 0$. Ainsi, la fonction F sépare le plan en deux demi-plans $\{(x, y) \mid F(x, y) \leqslant \frac{1}{2}\}$ et

$\{(x,y) \mid F(x,y) \geqslant \frac{1}{2}\}$ le long de la droite $ax+by+c=0$.



- L'erreur commise par la fonction F associée aux poids a, b, c est :

$$E = E(a,b,c) = \frac{1}{N}\sum_{i=1}^{N} E_i(a,b,c)$$

où N est le nombre total des données (le nombre de ronds bleus plus le nombre de carrés rouges) et

$$E_i(a,b,c) = \big(F(x_i,y_i) - z_i\big)^2.$$

On peut détailler un peu plus pour chaque type de point. En effet, pour un rond bleu la sortie attendue est $z_i = 0$ donc $E_i(a,b,c) = \big(\sigma(ax_i+by_i+c)-0\big)^2$, alors que pour un carré rouge la sortie attendue est $z_i = 1$ donc $E_i(a,b,c) = \big(\sigma(ax_i+by_i+c)-1\big)^2$.

1.2. Descente de gradient

- Pour trouver les poids $P = (a_1, a_2, \ldots)$ qui définissent le meilleur réseau $\mathcal{R}$ (autrement dit la meilleure fonction F), il suffit de minimiser l'erreur E, vue comme une fonction des poids $P = (a_1, a_2, \ldots)$. Pour cela on utilise la méthode de la descente de gradient.
- On part de poids initiaux $P_0 = (a_1, a_2, \ldots)$, par exemple choisis au hasard. On fixe un pas δ.
- On construit par itérations des poids P_k selon la formule de récurrence :

$$P_{k+1} = P_k - \delta \operatorname{grad} E(P_k).$$

À chaque itération, l'erreur $E(P_k)$ diminue. On s'arrête au bout d'un nombre d'itérations fixé à l'avance.
- Pour calculer le gradient $\operatorname{grad} E = \frac{1}{N}\sum_{i=1}^{N} \operatorname{grad} E_i$, il faut calculer chacun des $\operatorname{grad} E_i$, c'est-à-dire les dérivées partielles par rapport à chacun des poids a_j selon la formule :

$$\frac{\partial E_i}{\partial a_j}(X_i) = 2\frac{\partial F}{\partial a_j}(X_i)\big(F(X_i) - z_i\big).$$

Exemple.
Poursuivons l'étude de notre exemple.

- La fonction $E(a,b,c)$ dépend des poids a, b, c.
- On part de poids initiaux $P_0 = (a_0, b_0, c_0)$, par exemple $P_0 = (0, 1, -2)$ qui correspond à séparer le plan selon la droite horizontale $y = 2$. On fixe le pas $\delta = 1$.
- On calcule l'erreur locale pour la donnée numéro i :

$$E_i(a,b,c) = \big(F(x_i,y_i) - z_i\big)^2 = \big(\sigma(ax_i+by_i+c) - z_i\big)^2$$

avec $z_i = 0$ ou $z_i = 1$.

- Comme $\sigma'(x) = \sigma(x)(1-\sigma(x))$, alors, en notant $\sigma_i = \sigma(ax_i + by_i + c)$, on a :

$$\frac{\partial E_i}{\partial a}(a,b,c) = 2x_i\sigma_i(1-\sigma_i)(\sigma_i - z_i)$$
$$\frac{\partial E_i}{\partial b}(a,b,c) = 2y_i\sigma_i(1-\sigma_i)(\sigma_i - z_i)$$
$$\frac{\partial E_i}{\partial c}(a,b,c) = 2\sigma_i(1-\sigma_i)(\sigma_i - z_i)$$

et par suite

$$\operatorname{grad} E_i(a,b,c) = \left(\frac{\partial E_i}{\partial a}(a,b,c), \frac{\partial E_i}{\partial b}(a,b,c), \frac{\partial E_i}{\partial c}(a,b,c)\right) \qquad \text{et} \qquad \operatorname{grad} E(a,b,c) = \frac{1}{N}\sum_{i=1}^{N} \operatorname{grad} E_i.$$

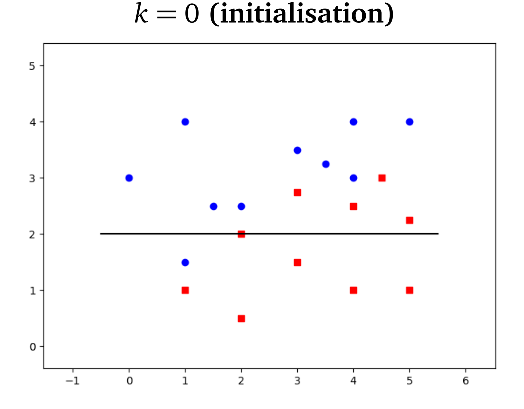
- On part de $P_0 = (0,1,-2)$, on calcule $\operatorname{grad} E(P_0) = (-0.077, 0.192, 0.005)$. On obtient le poids suivant par $P_1 = P_0 - \delta \operatorname{grad} E(P_0) = (0.077, 0.807, -2.005)$ (avec $\delta = 1$).
- On continue avec la formule de récurrence $P_{k+1} = P_k - \delta \operatorname{grad} E(P_k)$ pour obtenir les poids suivants :

k	$P_k = (a_k, b_k, c_k)$	$\operatorname{grad} E(a_k, b_k, c_k)$	$E(a_k, b_k, c_k)$
0	$(0,1,-2)$	$(-0.077, 0.192, 0.005)$	0.450
1	$(0.077, 0.807, -2.005)$	$(-0.072, 0.213, 0.0069)$	0.404
2	$(0.149, 0.593, -2.012)$	$(-0.094, 0.188, -0.0062)$	0.355
3	$(0.244, 30.4, -2.006)$	$(-0.120, 0.137, -0.0237)$	0.313
4	$(0.364, 0.267, -1.982)$	$(-0.090, 0.126, -0.0240)$	0.282
5	$(0.455, 0.140, -1.958)$	$(-0.073, 0.109, -0.0263)$	0.259
6	$(0.528, 0.031, -1.932)$	$(-0.059, 0.095, -0.0278)$	0.242
7	$(0.588, -0.063, -1.904)$	$(-0.051, 0.083, -0.0290)$	0.229
8	$(0.639, -0.146, -1.875)$	$(-0.044, 0.074, -0.0297)$	0.219
9	$(0.684, -0.221, -1.845)$	$(-0.040, 0.067, -0.0302)$	0.211
10	$(0.72, -0.288, -1.815)$	$(-0.036, 0.061, -0.0305)$	0.205
...			
100	$(1.328 - 1.828, 0.333)$	$(-0.00037, 0.00646, -0.01780)$	0.103
...			
1000	$(2.032, -3.860, 3.703)$	$(-0.00034, 0.00073, -0.00086)$	0.071

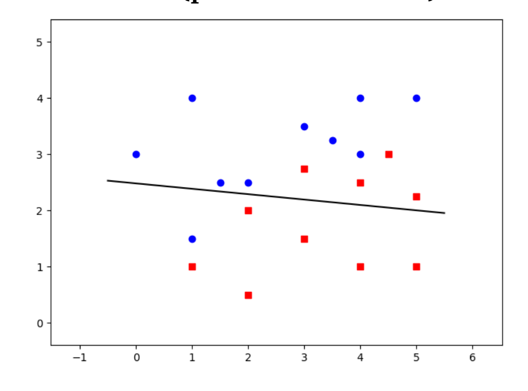
Au bout de $k = 1000$ itérations, on obtient les poids $P_{1000} = (2.032, -3.860, 3.703)$. Le gradient est devenu très petit, ce qui signifie ici que l'on est proche d'un minimum. De plus, l'erreur ne diminue presque plus lors des itérations suivantes, nous avons atteint le minimum recherché. (Attention, l'erreur est très petite, mais ne tend pas vers 0.)

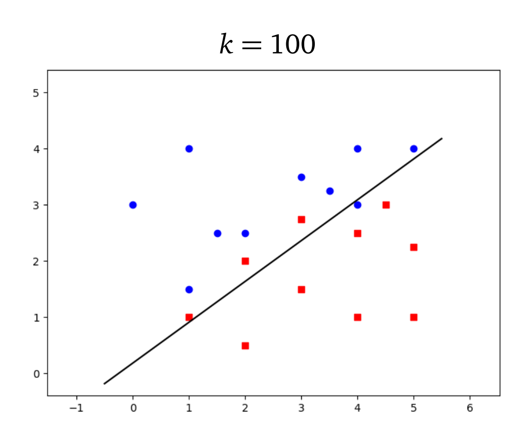
- La droite $ax + by + c = 0$ qui sépare le plan en deux demi-plans $\{(x,y) \mid F(x,y) \leqslant \frac{1}{2}\}$ et $\{(x,y) \mid F(x,y) \geqslant \frac{1}{2}\}$ évolue à chaque itération pour finir par séparer le mieux possible les ronds des carrés.

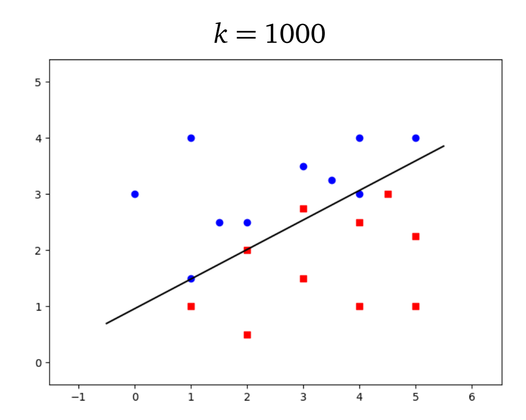
$k = 0$ (initialisation)



$k = 1$ (première itération)





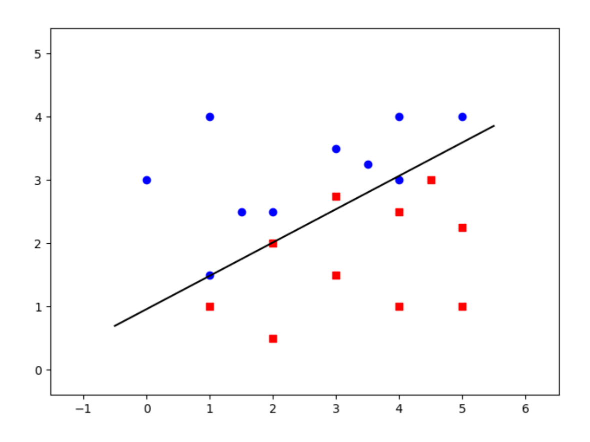


1.3. Prédiction

La conception d'un réseau de neurones est réalisée en modélisant au mieux les données injectées. Mais l'objectif réel est de faire des prédictions pour de nouvelles valeurs, jamais rencontrées auparavant. La descente de gradient produit un ensemble de poids P qui définit complètement notre réseau $\mathcal{R}$. Nous obtenons donc une fonction $F : \mathbb{R}^n \to \mathbb{R}^p$, construite de sorte que $F(X_i) \simeq z_i$. Nous pouvons évaluer cette fonction pour tout $X \in \mathbb{R}^n$, même pour des X différents des X_i.

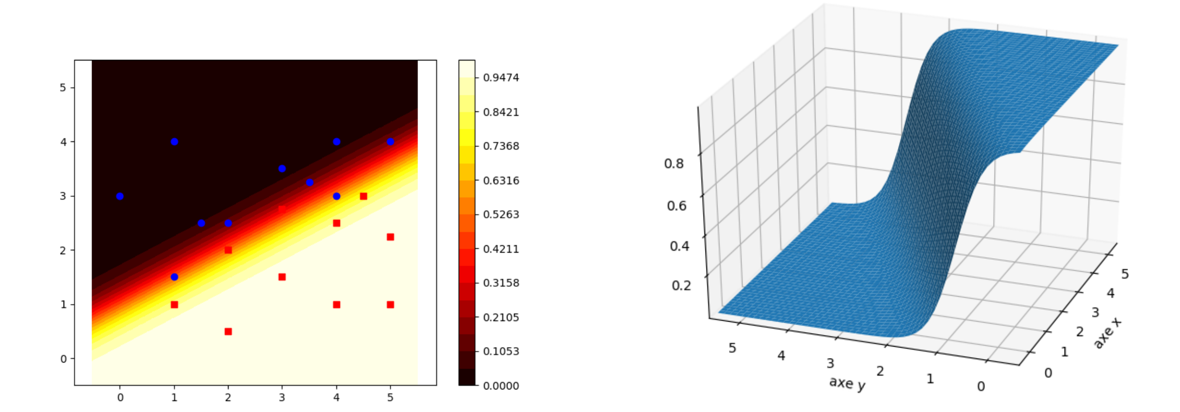
Exemple.
Dans notre exemple, nous avons obtenu par la descente de gradient les poids $P_{1000} = (a, b, c) = (2.032, -3.860, 3.703)$. Notre neurone $\mathcal{R}$ est maintenant opérationnel et définit ici la fonction

$$F(x, y) = \sigma(ax + by + c).$$



On souhaitait avoir $F(x_i, y_i) = 0$ pour les ronds bleus et $F(x_i, y_i) = 1$ pour les carrés rouges. Dans la pratique, on obtient des valeurs approchées, par exemple $F(0,3) = 0.0003$ pour le rond bleu en $(0,3)$ et $F(1,1) = 0.86$ pour le carré rouge en $(1,1)$.
Notre fonction F ne modélise pas parfaitement toutes nos données. Par exemple, pour le rond bleu en $(4,3)$ on a $F(4,3) = 0.56$ alors qu'on voudrait une valeur proche de 0 et de même pour le carré rouge en $(3, 2.75)$ on a $F(3, 2.75) = 0.30$ alors qu'on voudrait une valeur proche de 1.
Mais l'intérêt principal de F est d'être définie pour tous les points du plan, ceci permet d'attribuer une couleur à chaque point $(x, y) \in \mathbb{R}^2$ selon la convention : bleu si $F(x, y) \simeq 0$, rouge si $F(x, y) \simeq 1$. Il y a bien sûr une « zone grise » entre les zones rouge et bleue.
Par exemple, $F(2,3) = 0.02$ donc $(2,3)$ mérite d'être colorié en bleu, $F(2,1) = 0.98$ donc $(2,1)$ mérite d'être colorié en rouge. La frontière en laquelle F vaut $\frac{1}{2}$ est la droite d'équation $ax + by + c = 0$.

Voici les niveaux de F dans le plan (à gauche) et le graphe de F dans l'espace (à droite).

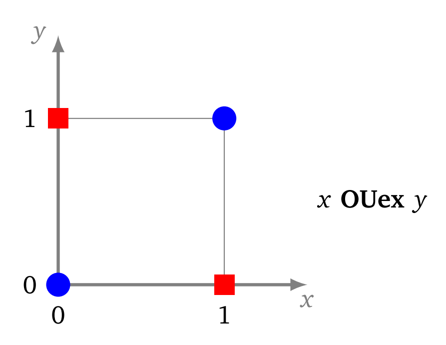


Nous avons choisi un réseau avec un seul neurone, la fonction F est donc nécessairement très simple (quels que soient les poids) et ne peut pas « coller » parfaitement aux données. Nous avons séparé au mieux les ronds bleus des carrés rouges par une droite. Avec un réseau plus complexe, et donc une frontière plus compliquée, nous aurions pu « coller » parfaitement aux données. Cependant est-ce vraiment ce que nous souhaitons faire ? Pour les données de notre exemple, avoir une séparation par une droite semble raisonnable. Peut-être que les points transfuges sont des erreurs de mesure.

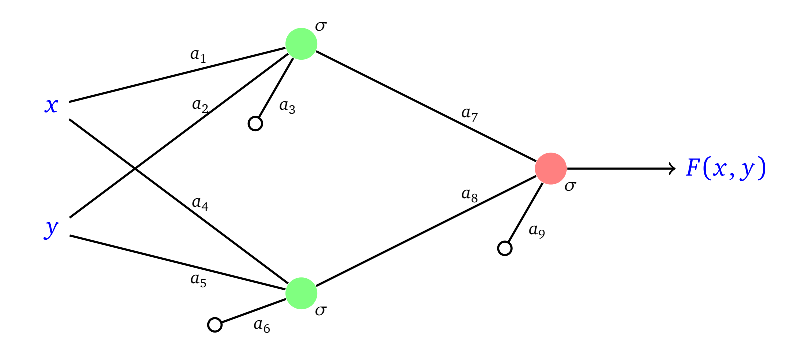
2. Exemples

2.1. Le « ou exclusif »

Rappelons le problème du « ou exclusif » : il s'agit de trouver un réseau de neurones dont la fonction associée F vaut 1 en $(0,1)$ et $(1,0)$ (les carrés rouges) et vaut 0 en $(0,0)$ et $(1,1)$ (les rond bleus).



Réseau.
On a vu qu'il existe une solution avec un réseau à deux couches de fonction d'activation la fonction de Heaviside (voir le chapitre « Réseau de neurones »). On cherche à l'aide de la machine quels poids pourraient convenir avec la fonction d'activation sigmoïde :



Il y a donc 9 coefficients à déterminer. La fonction d'erreur est :

$$E = (F(0,0)-0)^2 + (F(1,1)-0)^2 + (F(0,1)-1)^2 + (F(1,0)-1)^2$$

sachant que F et donc l'erreur E dépendent des coefficients $a_1, \ldots, a_9$. Il s'agit de trouver ceux qui rendent l'erreur E minimale.

Descente de gradient. On applique la méthode de la descente de gradient à la fonction E pour les variables $(a_1, \ldots, a_9)$. On utilise la descente de gradient classique avec un pas $\delta = 1$. Le choix des poids initiaux est déterminant pour les résultats. On choisit comme poids de départ :

$$P_0 = (a_1, \ldots, a_9) = (1.0, 2.0, -3.0, -3.0, -2.0, 1.0, 1.0, 1.0, -1.0).$$

L'erreur initiale vaut $E(P_0) = 0.2961$. Le gradient $\operatorname{grad} E(P_0)$ vaut

$$(0.00349, -0.00303, -0.00832, -0.00696, -0.01350, -0.01047, -0.00324, 0.01258, -0.04671).$$

Avec $\delta = 1$, la première itération modifie un petit peu les coefficients :

$$P_1 = (0.9965, 2.0030, -2.9916, -2.9930, -1.9864, 1.0104, 1.0032, 0.9874, -0.9532),$$

l'erreur a légèrement diminué et vaut maintenant $E(P_1) = 0.2935$. La seconde itération donne :

$$P_2 = (0.9925, 2.0055, -2.9839, -2.9860, -1.9731, 1.0206, 1.0053, 0.9738, -0.9105)$$

et l'erreur vaut maintenant $E(P_2) = 0.2911$.
Au bout de 1000 itérations :

$$P_{1000} = (3.5623, 3.5675, -5.5344, -5.3682, -5.3935, 1.8403, -6.3234, -6.3801, 3.1216)$$

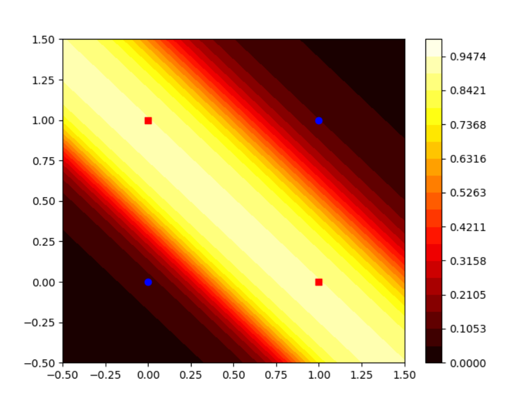
et l'erreur vaut maintenant $E(P_{1000}) = 0.0097$.

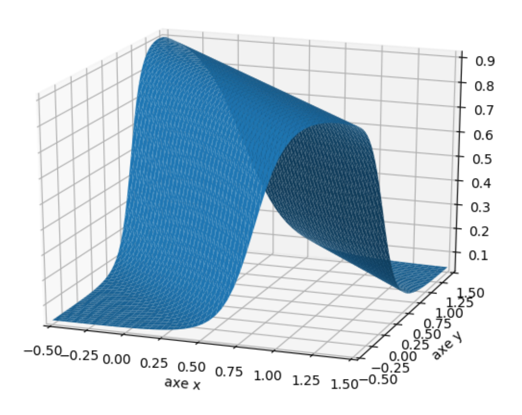
Validation et prédiction. Au bout de 1000 itérations notre réseau de neurones est assez performant. La fonction $F(x, y)$ obtenue vérifie :

$$F(0,0) = 0.08, \qquad F(1,1) = 0.10, \qquad F(1,0) = 0.89, \qquad F(0,1) = 0.89.$$

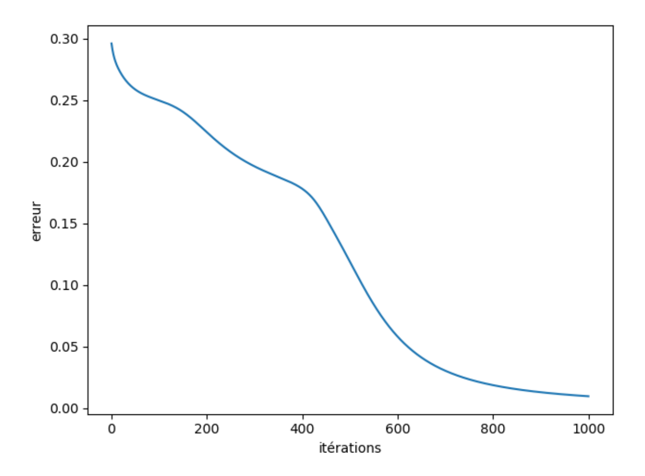
Les ronds bleus sont donc clairement séparés des carrés rouges par les valeurs de F.
Quelle couleur est-il naturel d'associer au points $(0.2, 0.9)$? On calcule $F(0.2, 0.9) = 0.87$ qui est assez proche de 1, il est donc naturel de le colorier en rouge.

Visualisation graphique. Voici deux représentations graphiques de la fonctions F obtenue. À gauche par les lignes de niveau dans le plan et à droite par son graphe dans l'espace. La fonction F prend des valeurs proches de 1 autour de la droite passant par $(1, 0)$ et $(0, 1)$ et se rapproche de 0 partout ailleurs.





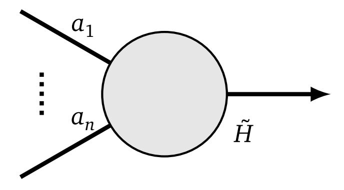
Analyse. Voici l'évolution de l'erreur en fonction du nombre d'itérations. L'erreur décroît au fil des itérations : c'est le principe de la descente de gradient !



Comparer les poids obtenus avec ceux du « ou exclusif » définis dans le chapitre « Réseau de neurones ».

2.2. Le perceptron et la règle de Hebb

Nous revenons sur l'exemple historique du cas d'un seul neurone (sans biais). Que donne la rétropropagation dans ce cas très simple ?



Nous souhaitons une fonction d'activation du type marche Heaviside, qui renvoie 0 ou 1, mais pour la descente du gradient nous avons besoin d'une dérivée non nulle. Nous allons imaginer qu'il existe une fonction marche de Heaviside virtuelle $\widetilde{H}$ telle que

$$\widetilde{H}(x) = 0 \quad \text{si } x < 0 \qquad \widetilde{H}(x) = 1 \quad \text{si } x \geqslant 0 \quad \text{et} \quad \widetilde{H}'(x) = 1 \quad \text{pour tout } x \in \mathbb{R}.$$

Il est clair qu'une telle fonction n'existe pas (car la dérivée d'une fonction constante par morceaux est toujours nulle) mais faisons comme si c'était le cas.
Ce neurone définit une fonction F telle que $F(x_1, \dots, x_n) = \widetilde{H}(a_1 x_1 + \cdots + a_n x_n)$. Comme d'habitude, nous avons des données (X_i, z_i), ici $z_i = 0$ ou bien $z_i = 1$. Il s'agit de trouver les poids $(a_1, \dots, a_n)$ tels que $F(X_i) \simeq z_i$. L'erreur locale est donnée par $E_i = \big(F(X_i) - z_i\big)^2$.

La descente de gradient (stochastique) pour la donnée i s'écrit :

$$P_{k+1} = P_k - \delta \operatorname{grad} E_i(P_k).$$

On a :

$$\frac{\partial E_i}{\partial a_j}(X_i) = 2\frac{\partial F}{\partial a_j}(X_i)\big(F(X_i) - z_i\big).$$

Mais $\frac{\partial F}{\partial a_j}(X_i) = x_j$ car $\widetilde{H}'(x) = 1$ et comme $F(X_i)$ et z_i valent 0 ou 1, alors $\frac{\partial E_i}{\partial a_j}(X_i)$ vaut $\pm 2x_j$ ou 0. Autrement dit, $\operatorname{grad} E_i = 2\epsilon(x_1,\ldots,x_n) = 2\epsilon X_i$ avec $\epsilon = 0$, 1 ou -1.

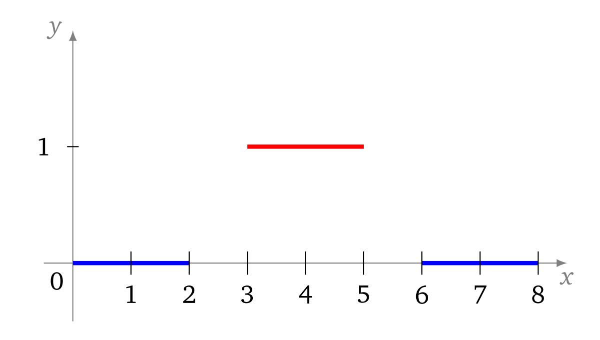
Nous avons ainsi obtenu la ***règle de Hebb***, qui est en fait l'ancêtre de la rétropropagation.

Règle de Hebb.

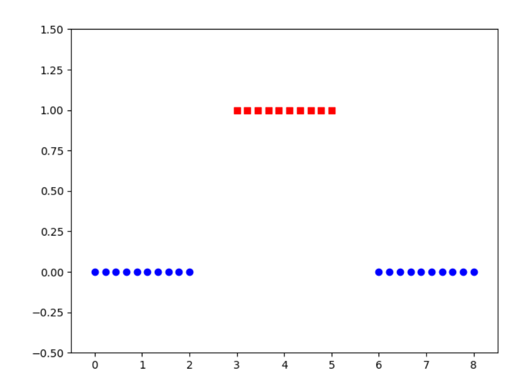
- On fixe un pas $\delta > 0$.
- On part d'un poids P_0 (choisi au hasard par exemple).
- On calcule par récurrence les poids P_k en parcourant la liste (X_i, y_i) et en distinguant les cas suivants :
  — si la sortie prédite $F(X_i)$ vaut la sortie attendue z_i alors on ne change rien :
  $$P_{k+1} = P_k,$$
  — si la sortie prédite $F(X_i)$ vaut 1 alors que la sortie attendue z_i vaut 0 alors :
  $$P_{k+1} = P_k - 2\delta X_i,$$
  — si la sortie prédite $F(X_i)$ vaut 0 alors que la sortie attendue z_i vaut 1 alors :
  $$P_{k+1} = P_k + 2\delta X_i.$$
- Une fois toutes les données (X_i, z_i) utilisées, on recommence depuis la première donnée.
- On s'arrête au bout d'un nombre d'itérations fixé à l'avance.
- Le dernier poids obtenu $P_k = (a_1,\ldots,a_n)$ définit le perceptron et la fonction associée qui répond au problème est : $F(x_1,\ldots,x_n) = 1$ si $a_1x_1 + \cdots + a_nx_n \geqslant 0$ et $F(x_1,\ldots,x_n) = 0$ sinon.

2.3. Une fonction marche

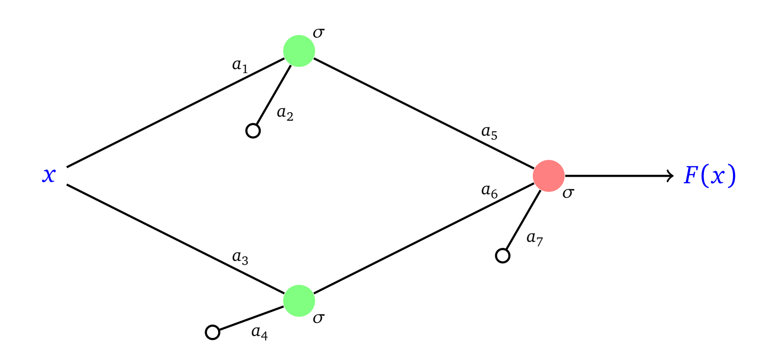
Nous avons vu dans le chapitre « Réseau de neurones » qu'il est facile de réaliser des fonctions « marche » à l'aide d'un réseau simple et de la fonction de Heaviside. Nous souhaitons construire un réseau de neurones dont la fonction $x \mapsto F(x)$ réalise au mieux les contraintes suivantes : $F(x)$ vaut 0 sur $[0,2]$, puis vaut 1 sur $[3,5]$ et de nouveau vaut 0 sur $[6,8]$ (il y a une certaine liberté sur $[2,3]$ et $[5,6]$).



Données. On décide de placer 10 ronds bleus espacés régulièrement sur $[0,2]$, la même chose sur $[6,8]$, là où la fonction doit être nulle, et également 10 carrés rouges espacés régulièrement sur $[3,5]$ là où la fonction doit valoir 1. Ces points fournissent donc les $N = 30$ données d'apprentissage.



Réseau. On décide d'utiliser un réseau avec deux neurones sur la couche d'entrée et un neurone sur la couche de sortie, tous utilisant la fonction d'activation σ.



Il y a 7 coefficients à déterminer. La fonction d'erreur E est la somme des $(F(x_i)-0)^2$ pour les x_i abscisses des ronds bleus et des $(F(x_i)-1)^2$ pour les x_i abscisses des carrés rouges.

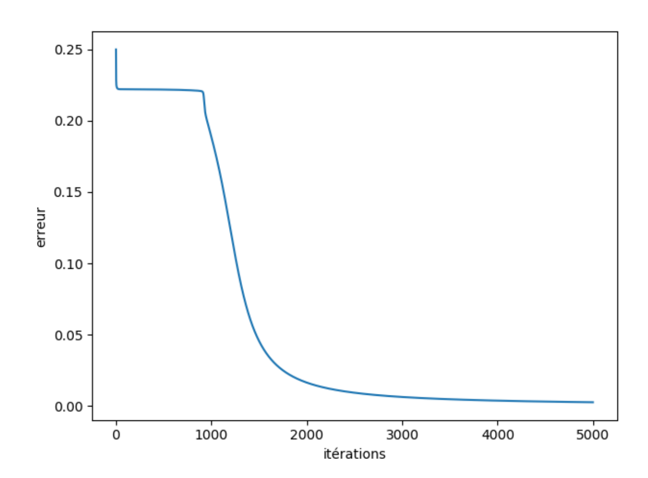
Descente de gradient. On applique la descente de gradient à la fonction E de variables $(a_1,\ldots,a_7)$ pour un pas $\delta=1$ et un choix arbitraire de poids initiaux :

$$P_0=(a_1,\ldots,a_7)=(0.0,1.0,0.0,-1.0,1.0,1.0,-1.0).$$

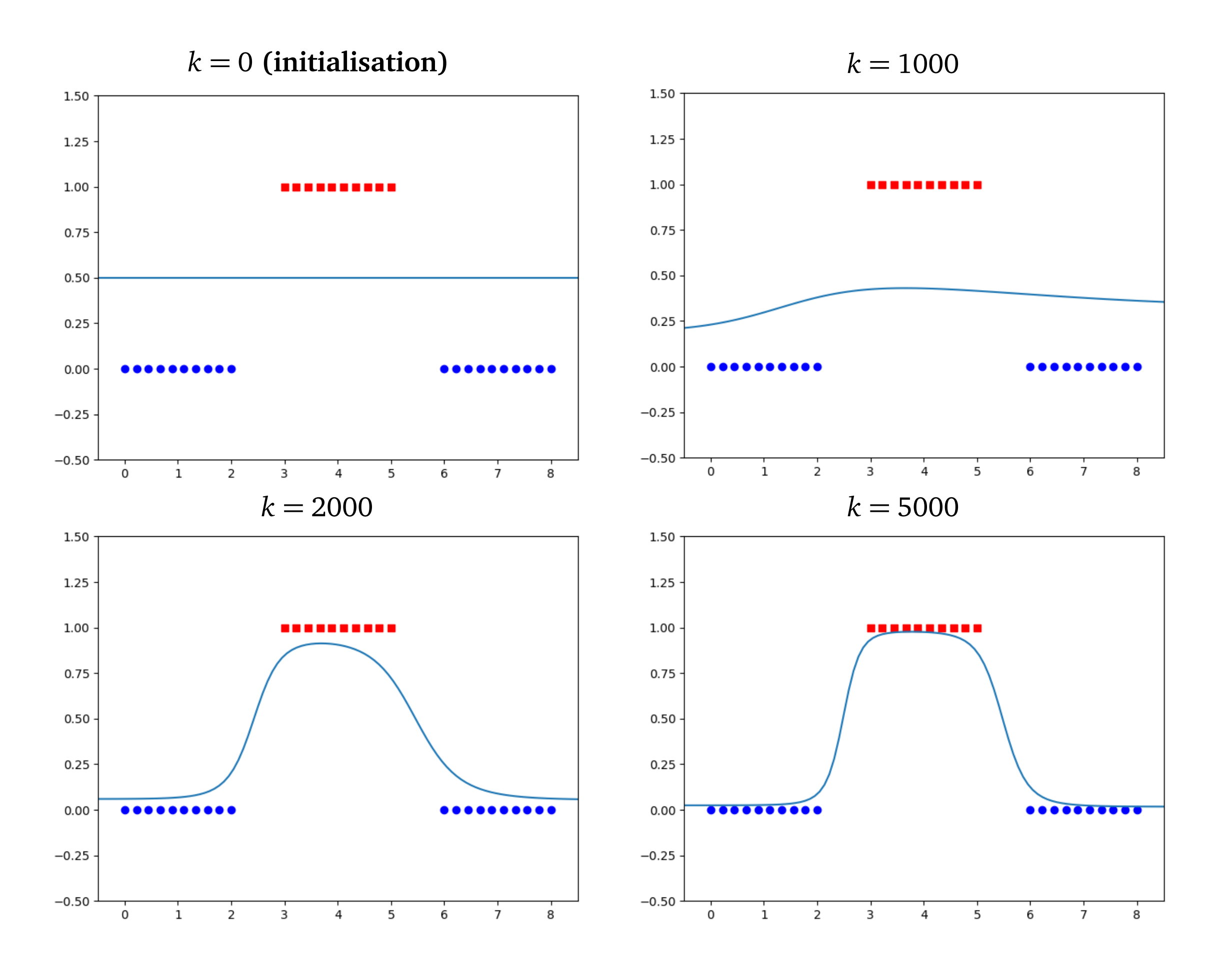
Au bout de 5000 itérations, on obtient les poids :

$$P_{5000}=(a_1,\ldots,a_7)=(-2.0142,10.9964,3.0543,-7.7086,8.2334,7.8735,-11.9037).$$

Voici l'évolution de l'erreur en fonction du nombre d'itérations.

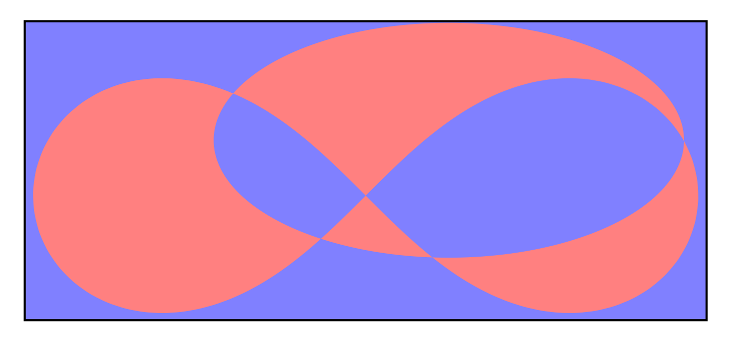


Visualisation graphique.
Voici le graphe de la fonction $x\mapsto F(x)$ au bout de différents nombres d'itérations.

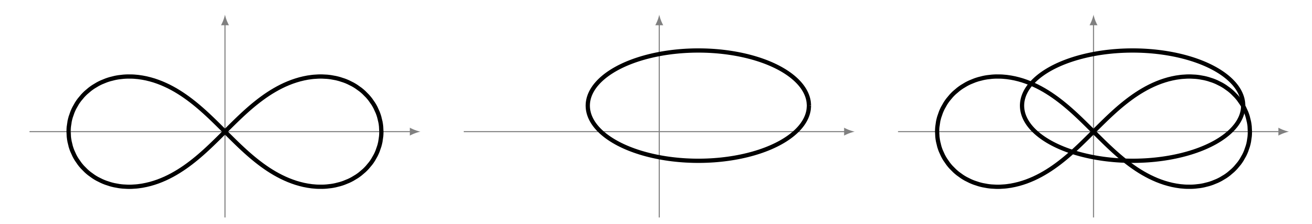


2.4. Plus de neurones

Le but est ici de trouver un réseau de neurones qui distingue deux zones compliquées du plan : la zone bleue et la zone rouge.



Voici comme sont construites ces zones : on part d'une lemniscate de Bernoulli d'équation $(x^2+y^2)^2 = 4(x^2-y^2)$ et d'une ellipse d'équation $(x-\frac{1}{2})^2+4(y-\frac{1}{3})^2 = 2$.

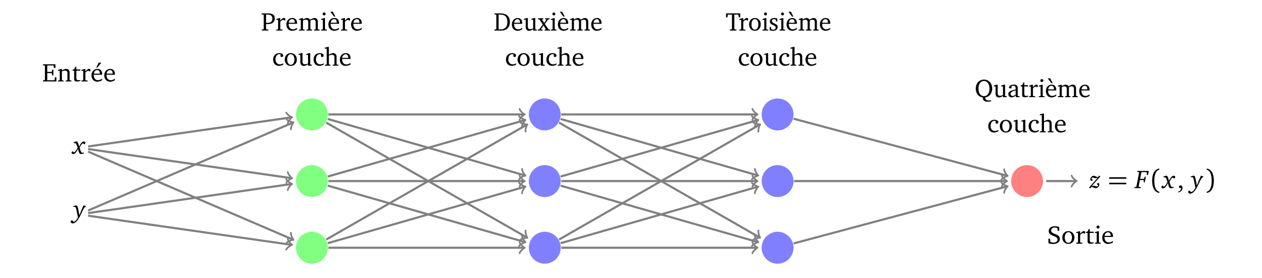


L'union de ces deux courbes a pour équation :

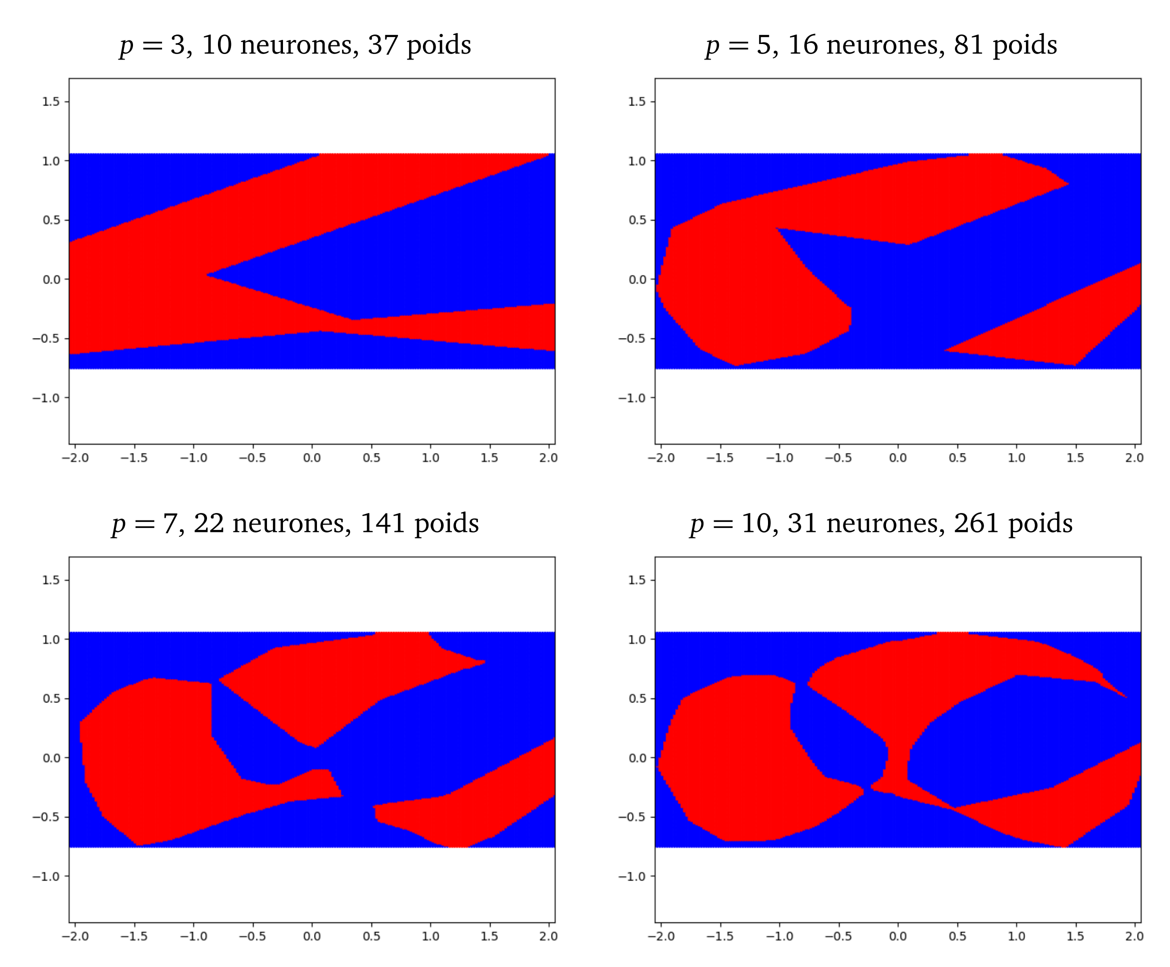
$$f(x,y) = \big((x^2+y^2)^2-4(x^2-y^2)\big)\cdot\big((x-\tfrac{1}{2})^2+4(y-\tfrac{1}{3})^2-2\big) = 0.$$

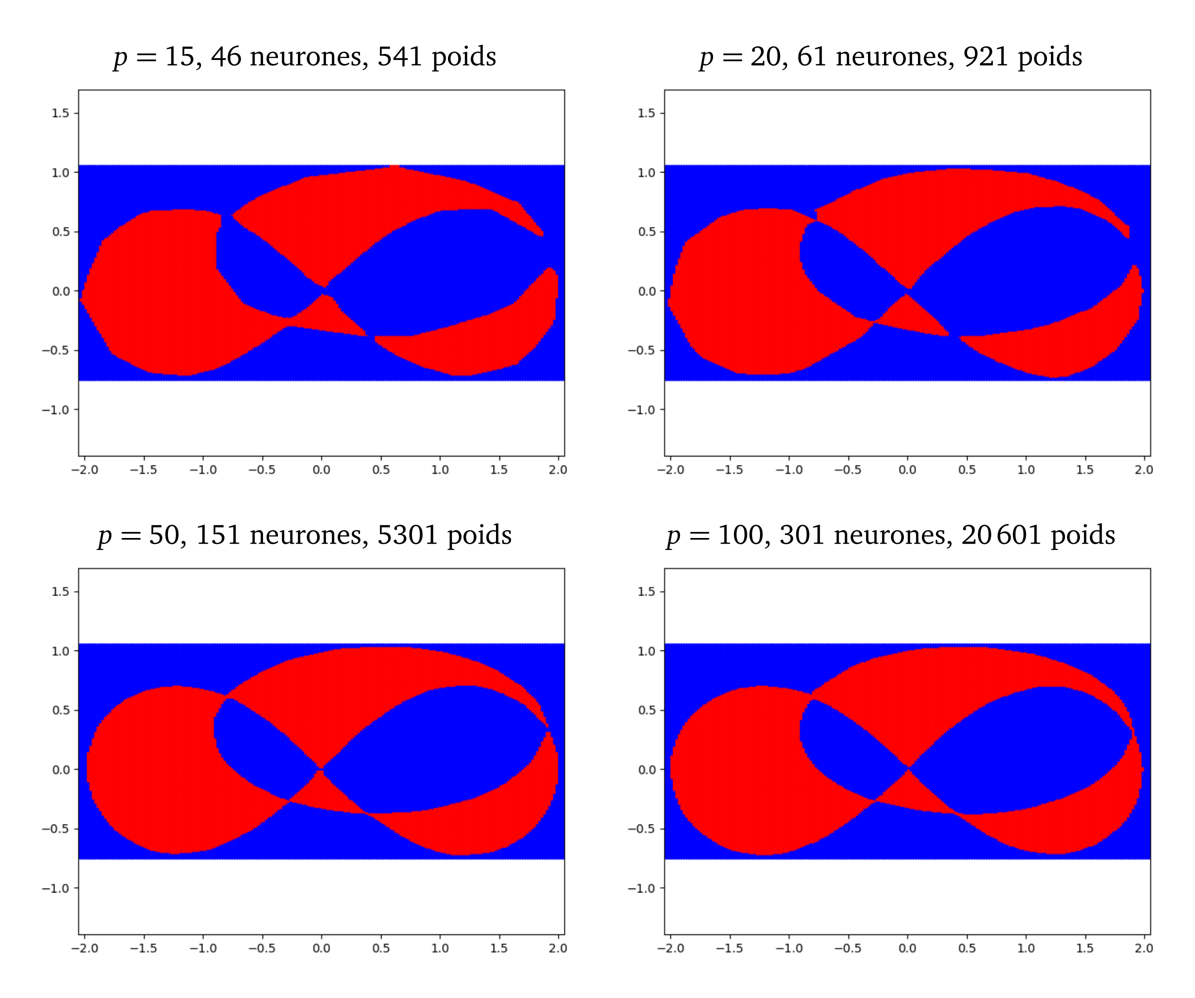
La zone rouge correspond aux points de coordonnées (x,y) pour lesquels $f(x,y) \leqslant 0$ et la zone bleue à $f(x,y) > 0$. Nous allons limiter notre étude à une zone rectangulaire. Le rectangle est choisi de sorte que l'aire bleue et l'aire rouge soient à peu près égales.

Objectifs. On oublie maintenant la fonction f et on ne retient que quelques points rouges et quelques points bleus. On cherche un réseau et une fonction F telle que $F(x, y) \simeq 1$ pour les points rouges et $F(x, y) \simeq 0$ pour les points bleus.
Données. On divise notre rectangle en une grille de $n \times n$ points. Les exemples ci-dessous sont calculés pour $n = 200$. Nous avons donc 40 000 points (x, y), repartis (à peu près) équitablement entre rouge ($z = 1$) et bleu ($z = 0$).
Architecture. On décide de concevoir un réseau à 4 couches, avec le même nombre p de neurones par couche pour les trois premières couches et un seul neurone sur la couche de sortie. La fonction d'activation choisie pour tous les neurones est ReLU . Voici une illustration de la configuration pour $p = 3$.



Poids. On calcule les poids avec une méthode de descente de gradient stochastique. Les poids initiaux sont choisis aléatoirement, le nombre d'itérations est suffisamment grand. Le réseau obtenu après calculs fournit donc une fonction F. On colorie en rouge les points pour lesquels $F(x, y) \simeq 1$ (en fait, là où $F(x, y) \geqslant \frac{1}{2}$) et en bleu les points pour lesquels $F(x, y) \simeq 0$ (en fait, là où $F(x, y) < \frac{1}{2}$). On regarde si le résultat obtenu est proche de la situation envisagée. On présente ici quelques résultats typiques intéressants (les résultats pouvant varier selon le choix aléatoire des poids initiaux).



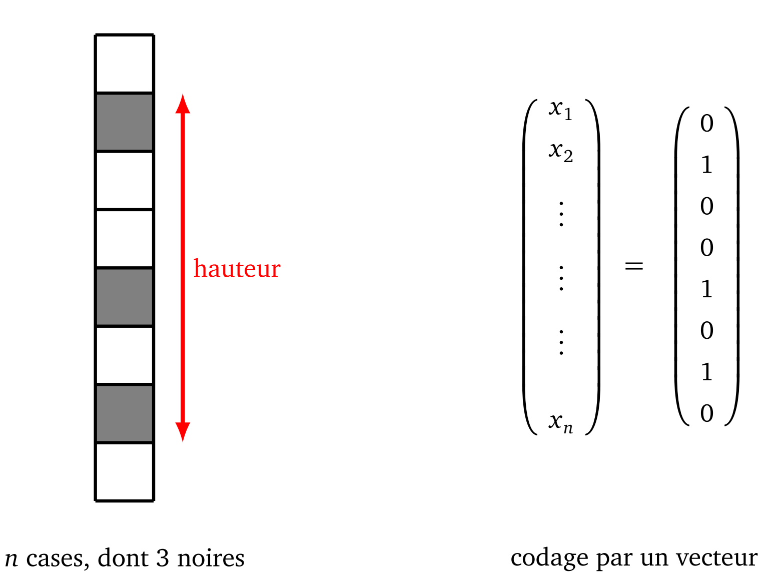


Conclusion : il faut un nombre assez grand de neurones pour pouvoir modéliser des phénomènes compliqués. Dans l'exemple traité ici, on obtient une bonne approximation à partir de $p = 20$, soit plus de 60 neurones et environ 1000 poids.

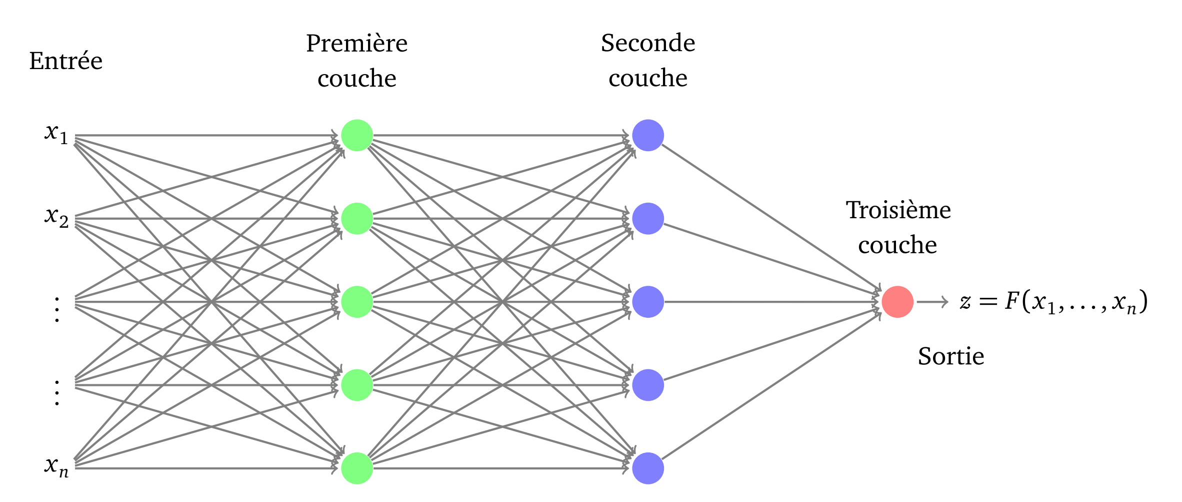
2.5. Apprentissage

N'oublions pas le but principal des réseaux de neurones : apprendre à partir des données afin de prédire des résultats pour des situations nouvelles.

Voici un problème pour lequel nous allons tester si les prédictions sur des données nouvelles sont correctes ou pas. On dispose de n cases, dont 3 sont noircies. On appelle *hauteur*, le nombre total de cases entre la plus haute et la plus basse case noircie. Si les trois cases ont pour rang $i < j < k$ alors la hauteur vaut $h = k - i + 1$ (la case noircie du milieu n'intervient pas dans la formule). Sur le dessin ci-dessous, $n = 8$ et la configuration représentée est de hauteur 6.



Nous voulons tester si un réseau de neurones permet de retrouver un résultat qu'on peut vérifier ici à l'aide d'une formule simple : étant donné une configuration il s'agit de faire calculer à la machine quelle est sa hauteur.
Il y a en tout $N = \frac{n(n-1)(n-2)}{6}$ configurations possibles pour les trois cases noires parmi n cases. On décide d'utiliser la moitié des configurations comme données d'apprentissage et l'autre moitié va nous permettre de tester si le réseau calculé avec les données d'apprentissage se comporte « correctement » sur les nouvelles données.
Dans la suite nous prendrons $n = 20$ cases, ce qui fait $N = 1140$ configurations possibles.
Données en entrées. On calcule les N configurations possibles. Après un mélange aléatoire, on ne retient que $N/2$ configurations pour l'apprentissage. Une configuration est codée sous la forme d'un vecteur $X = (x_1, x_2, \dots, x_n)$ avec $x_i = 0$ (case blanche) ou $x_i = 1$ (case noire). Pour chaque configuration X des données d'apprentissage, on calcule la hauteur $z = h(X)$. Les données d'apprentissage sont les $N/2$ données (X_k, z_k) formées des entrées X_k et de la sortie attendue z_k correspond à la hauteur.
Réseau. Le réseau comporte n entrées : une entrée par case. Il est constituée de 3 couches. Les deux premières couches possèdent p neurones et la couche de sortie un seul. La fonction d'activation est la fonction ReLU pour tous les neurones. Sur la figure ci-dessous est illustré le cas $p = 5$.



Apprentissage. Le réseau est initialisé avec des poids aléatoires. Ensuite ces poids sont modifiés par une descente de gradient stochastique avec un nombre suffisant d'itérations.
Sortie prédite. Le réseau paramétré fournit une fonction F à valeur réelle. La sortie attendue étant un entier, on décide que la sortie prédite sera arrondie à l'entier le plus proche. Pour chaque X_k, la prédiction est correcte si la valeur arrondie $F(X_k)$ vaut la hauteur $h(X_k)$.

Tests. Il faut tester l'efficacité du réseau : tout d'abord le réseau modélise-t-il correctement les données d'apprentissage ? En effet, même si la fonction F a été construite dans le but d'avoir $F(X_k) \simeq z_k$, il n'y a pas nécessairement égalité pour toutes les données d'apprentissage. Mais surtout, il faut tester si la fonction F donne de bonnes prédictions pour de nouvelles données. Nous disposons pour cela des $N/2$ données de test non utilisées lors de l'apprentissage.
Les résultats dépendent des poids initiaux (aléatoires) et de la liste des configurations choisies pour l'apprentissage (liste qui a été mélangée), on ne retient que les meilleurs résultats parmi plusieurs essais. Par exemple, pour $p = 10$, voici un des meilleurs résultats obtenus :

- *Apprentissage.* 560 données prédites correctement sur un total de de 570 données d'apprentissage, soit 98% de réussite.
- *Test.* 506 données prédites correctement sur un total de de 570 données de test, soit 89% de réussite.

Voici quelques résultats pour différentes valeurs de p.

p	neurones	poids	pourcentage apprentissage	pourcentage test
3	7	79	50%	45%
5	11	141	85%	75%
7	15	211	90%	85%
10	21	331	98%	90%
15	36	571	99%	90%
20	41	861	100%	85%

Commentaires. Plus il y a de neurones, plus la fonction F modélise correctement les données d'apprentissage : à partir de 20 neurones ($p \geqslant 10$), la modélisation est quasi-parfaite. Les pourcentages de réussite pour les données de test sont toujours inférieures et plafonnent à 90%. Un phénomène nouveau apparaît avec plus de 40 neurones (pour $p = 20$), les pourcentages de réussite sur les tests régressent par rapport à des réseaux ayant moins de neurones, bien que les données d'apprentissage soient parfaitement modélisées : c'est un phénomène de sur-apprentissage.

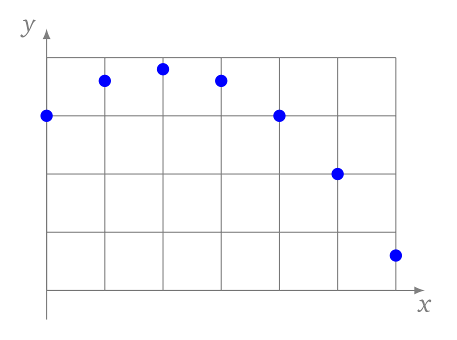
3. Sur-apprentissage et autres soucis

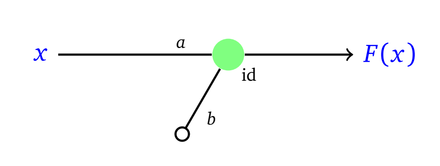
Nous allons voir différents problèmes qui peuvent intervenir dans le calcul des poids d'un réseau de neurones, le problème le plus subtil étant le *sur-apprentissage*.

3.1. Modèle insuffisant

Problème. Voici deux exemples de situations dans lesquelles le choix de l'architecture du réseau pose problème, car le réseau est trop simple.

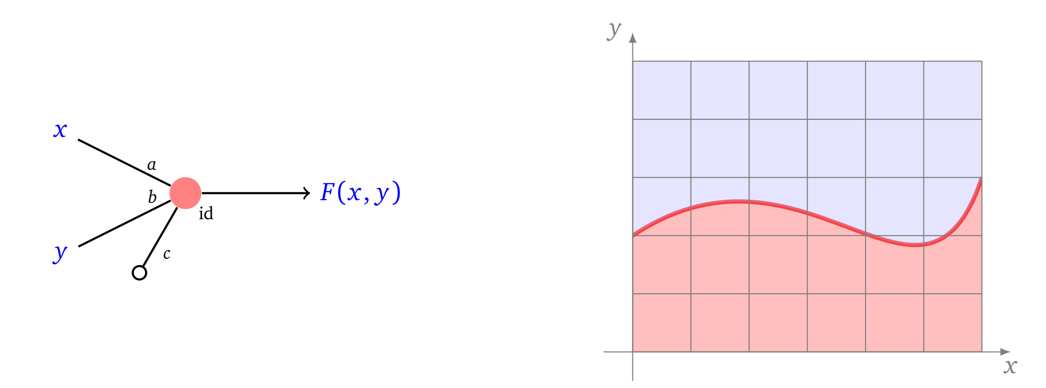
Exemple 1. Imaginons que nous voulions modéliser une situation à partir des données fournies par les points (x_i, y_i) du plan (figure de gauche). Si on construit un réseau d'un seul neurone (figure de droite), alors la fonction de sortie sera du type $y = F(x) = ax + b$.





Aucun paramètre (a, b) ne permettra une modélisation correcte de la situation, car les points ne sont franchement pas alignés.

Exemple 2. Le même problème se produirait si on voulait trouver une fonction F, construite à partir d'un seul neurone, valant 0 pour la zone bleue et 1 pour la zone rouge de la figure ci-dessous. Ce n'est pas possible car la frontière de la solution n'est pas linéaire.

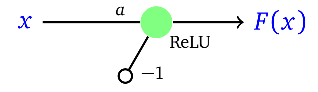


Conclusion. Dans ces situations, le problème ne se situe pas au niveau des poids, augmenter la taille des données ou le nombre d'itérations n'y changera rien. La conception du réseau est mauvaise pour répondre à la question posée. C'est comme vouloir faire une course de voitures avec une deux-chevaux. La solution est de changer l'architecture du réseau, par exemple en rajoutant des neurones.

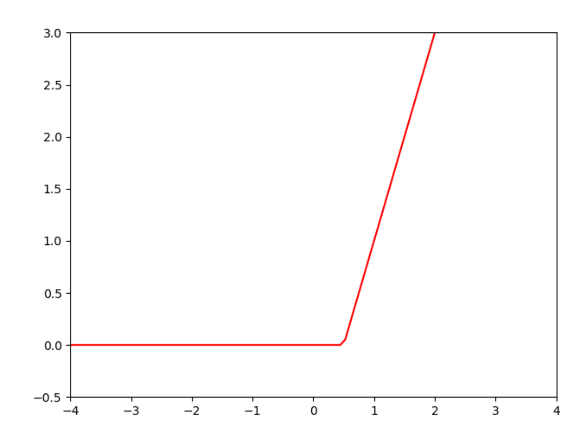
3.2. Minimum local

Problème. La descente de gradient a pour objectif de fournir un minimum local. Cependant rien n'affirme que ce minimum local est un minimum global.

Exemple. Voici un réseau composé d'un seul neurone de fonction d'activation ReLU et possédant un seul coefficient à déterminer, le biais étant imposé à la valeur -1.

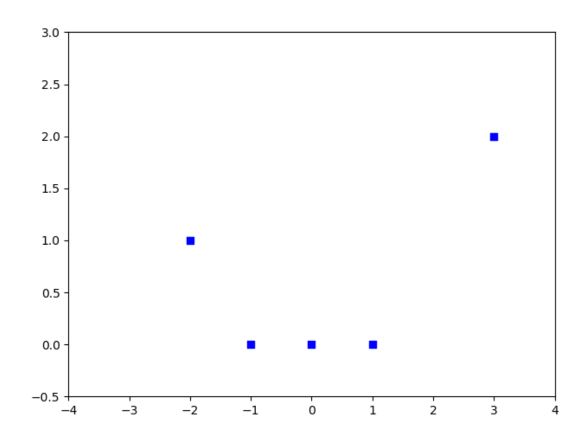


Le graphe de la fonction F correspondant à ce neurone est une portion de la droite d'équation $y = ax - 1$, prolongée par l'axe des abscisses. Voici l'exemple du graphe de F, avec $a = +2$.

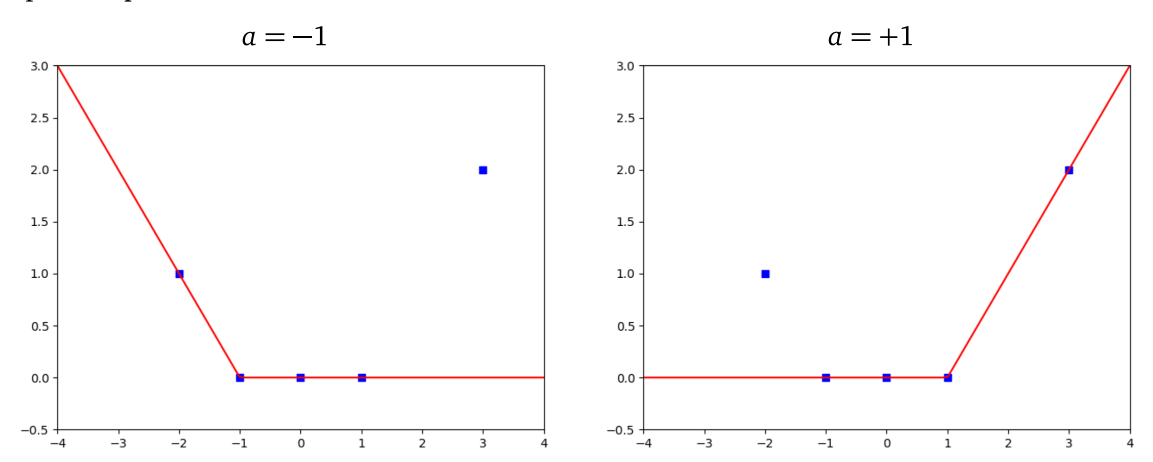


Les données sont des points déterminés par leurs coordonnées (x_i, y_i) :

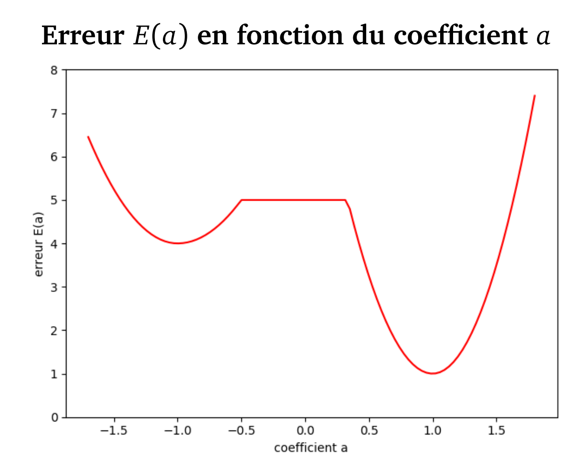
$$(-2, 1), \quad (-1, 0), \quad (0, 0), \quad (1, 0), \quad (3, 2).$$



Il s'agit de trouver le meilleur coefficient a, qui définisse F tel que $F(x_i) \simeq y_i$. Autrement dit, on souhaite minimiser $E(a) = \sum E_i(a)$ où $E_i(a) = (F(x_i) - y_i)^2$. Géométriquement il y a deux paramètres qui semblent meilleurs que les autres $a = -1$ (figure de gauche) et $a = +1$ (figure de droite) car alors les portions de droites passent par des carrés bleus.



Cette intuition se vérifie lorsque l'on trace le graphe de la fonction d'erreur $a \mapsto E(a)$. La fonction possède deux minimums locaux en $a = -1$ et $a = +1$ (et aussi une portion constante, due à l'usage de la fonction ReLU).



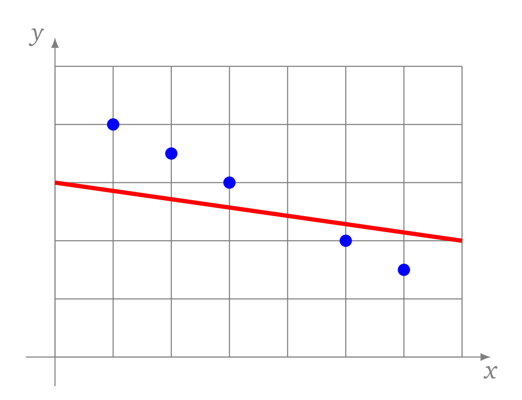
Le minimum en $a = +1$ est le minimum global. Mais si on applique la descente de gradient en partant d'un coefficient initial $a_0 < 0$, il y a de grandes chances d'arriver au minimum local $a = -1$, qui ne sera pas la solution optimale.

Conclusion. Vous êtes une fourmi qui se promène dans une boîte d'œufs avec des trous de différentes profondeurs : vous ne pouvez pas voir à l'avance quel est l'emplacement le plus profond ! Une solution peut être de tester différentes valeurs initiales, dans le but d'obtenir différents minimums locaux.

3.3. Sous-apprentissage

Problème. Le *sous-apprentissage* révèle une conception correcte de l'architecture du réseau mais une mauvaise mise en œuvre. On obtient alors des poids qui ne répondent pas correctement au problème.

Exemple. Une droite obtenue par un réseau approche mal une suite de points pourtant alignés.



Cela peut être dû aux raisons suivantes :

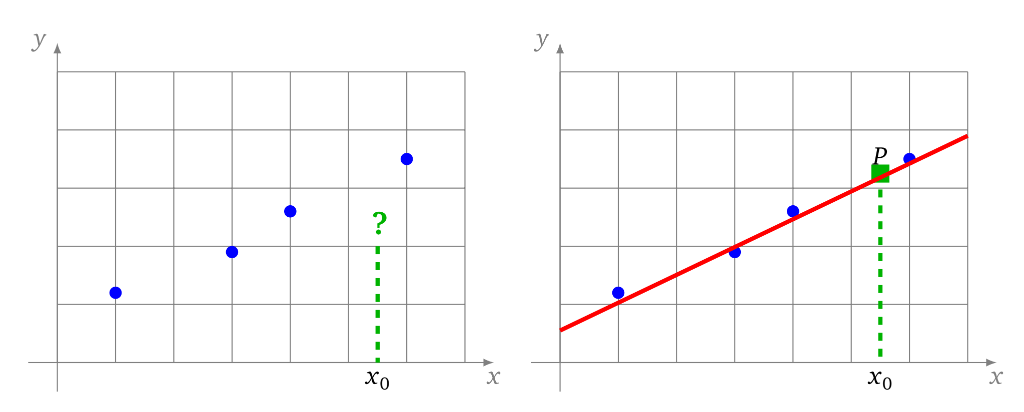
- les données ne sont pas en nombre suffisant,
- le nombre d'itérations est insuffisant,
- le pas δ est trop grand.

Conclusion. Il est difficile de savoir à l'avance quelle est la bonne taille des données à utiliser et combien d'itérations sont nécessaires pour arriver à un modèle correct. Le sous-apprentissage, c'est comme faire une course de voiture avec une porsche mais en ne dépassant jamais les 80 km/h ! A posteriori, la solution est simple : ajouter des données, augmenter le nombre d'itérations ou diminuer la pas.

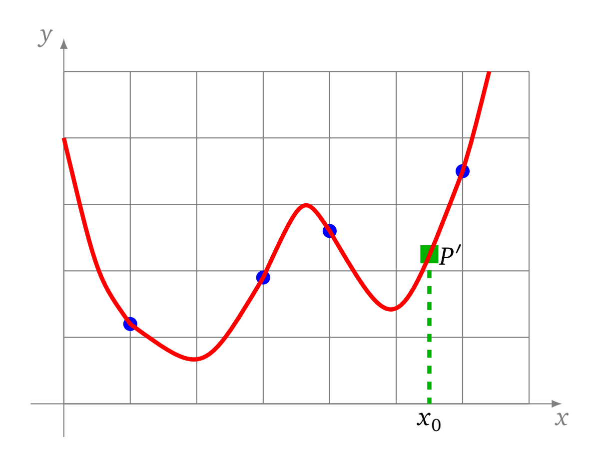
3.4. Sur-apprentissage

Problème. Le modèle obtenu « colle » parfaitement aux données d'apprentissage, mais cependant les prédictions pour de nouvelles valeurs sont mauvaises. Il s'agit donc d'un problème délicat : la fonction F obtenue vérifie bien $F(X_i) \simeq z_i$ pour toutes les données, mais pour une nouvelle entrée X, la sortie $F(X)$ n'est pas une bonne prédiction. Cela se produit lorsque l'on se concentre uniquement sur l'apprentissage à partir des données, mais que l'on a oublié que le but principal est la prédiction.

Exemple 1. Nous avons 4 points bleus. Pour une valeur x_0 donnée, on souhaite prédire une valeur y_0 (figure de gauche) cohérente avec nos données.
On propose dans un premier temps d'approcher la solution en utilisant une droite (figure de droite) même si les points ne sont pas exactement alignés. Cela permet de prédire une valeur y_0 pour placer le point $P = (x_0, y_0)$. Ce modèle ne sera jamais parfait car les points ne sont pas alignés, donc aucune droite ne convient exactement, autrement dit l'erreur ne sera jamais nulle.

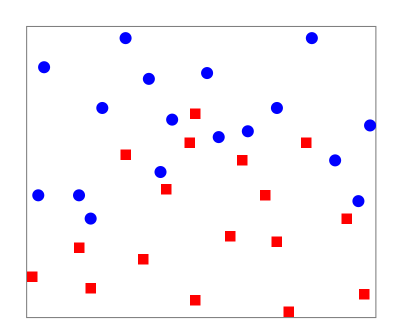
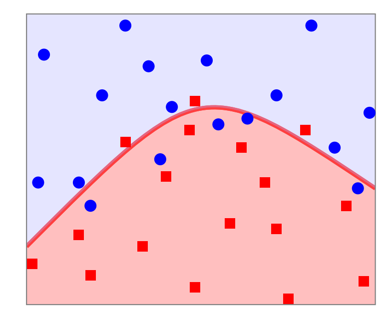
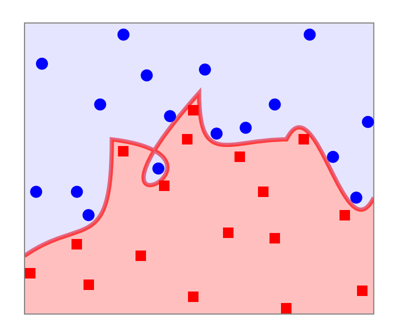


On peut construire un modèle plus compliqué, par exemple chercher une courbe polynomiale de degré 3 ou plus qui passe *exactement* par tous les points d'apprentissage. Ainsi, pour cette courbe, l'erreur sera nulle. Cette courbe permet de prédire une valeur y_0' pour placer un point $P'(x_0, y_0')$.



Ce dernier modèle répond entièrement au problème, mais la solution proposée ne semble pas raisonnable par rapport aux données de départ. C'est un cas de sur-apprentissage : le modèle est correct sur les données, mais les prédictions seront mauvaises.

Exemple 2. Le problème est similaire si on essaye de séparer les ronds bleus des carrés rouges de la figure de gauche ci-dessous. Une modèle simple permet à une parabole de séparer l'essentiel des ronds bleus des carrés rouges (figure centrale). On peut entraîner un modèle plus complexe pour qu'il délimite parfaitement les deux types de points, mais au prix d'une complexité non nécessairement voulue (figure de droite).

Conclusion. Le sur-apprentissage, c'est comme emmener ses enfants à l'école en conduisant à 200 km/h : cela répond de façon correcte à un problème, mais cela provoque beaucoup d'autres ennuis ! Quelle est la meilleure solution entre un modèle simple mais imparfait et un modèle parfait mais compliqué ? Il n'y a pas une réponse immédiate à la question : pour le savoir, il faut tester le modèle sur d'autres données, jamais rencontrées précédemment, afin de déterminer si le réseau n'a pas été sur-entraîné.

Python : tensorflow avec keras - partie 2

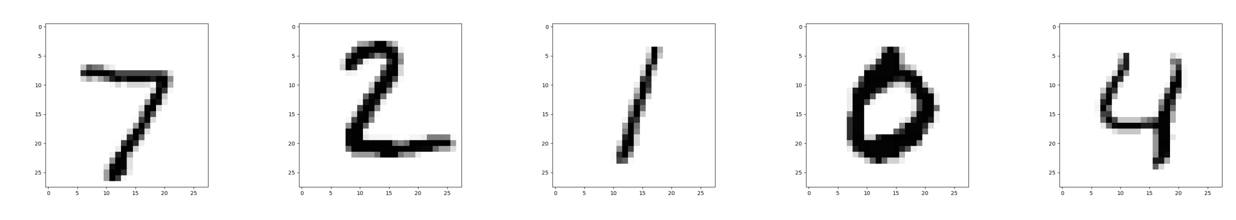
Chapitre 10

Vidéo ■ partie 10.1. Reconnaissance de chiffres
Vidéo ■ partie 10.2. Analyse de texte
Vidéo ■ partie 10.3. Reconnaissance d'images

Jusqu'ici nous avons travaillé dur pour comprendre en détails la rétropropagation du gradient. Les exemples que nous avons vus reposaient essentiellement sur des réseaux simples. En complément des illustrations mathématiques étudiées, il est temps de découvrir des exemples de la vie courante comme la reconnaissance d'image ou de texte. Nous profitons de la librairie tensorflow/keras *qui en quelques lignes nous permet d'importer des données, de construire un réseau de neurones à plusieurs couches, d'effectuer une descente de gradient et de valider les résultats.*

1. Reconnaissance de chiffres

Il s'agit de reconnaître de façon automatique des chiffres écrits à la main.

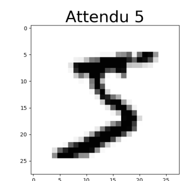


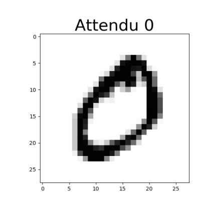
C'est l'un des succès historiques des réseaux de neurones qui permet par exemple le tri automatique du courrier par lecture du code postal.

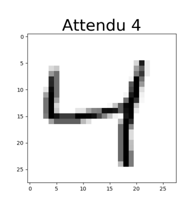
1.1. Données

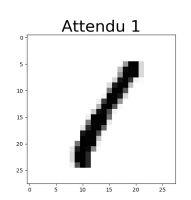
Base MNIST

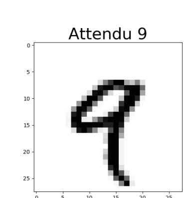
Un élément essentiel de l'apprentissage automatique est de disposer de données d'apprentissage nombreuses et de qualité. Une telle base est la base MNIST dont voici les premières images.



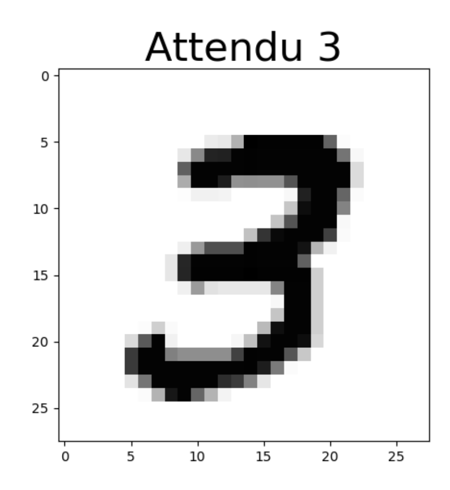








Une donnée est constituée d'une image et du chiffre attendu.



Plus en détails :

- La base est formée de 60 000 données d'apprentissage et de 10 000 données de test.
- Chaque donnée est de la forme : [une image, le chiffre attendu].
- Chaque image est de taille 28×28 pixels, chaque pixel contenant un des 256 niveaux de gris (numérotés de 0 à 255).

Ces données sont accessibles très simplement avec *tensorflow*/*keras* :

```
from tensorflow.keras.datasets import mnist
(X_train_data,Y_train_data),(X_test_data,Y_test_data) = mnist.load_data()
```

Il faut passer un peu de temps à comprendre et à manipuler les données. `X_train_data[i]` correspond à une image et `Y_train_data[i]` au chiffre attendu pour cette image. Nous parlerons plus tard des données de test. On renvoie au fichier `tf2_chiffres_data.py` pour une exploration de la base. Noter que l'on peut afficher une image à l'aide de *matplotlib* par la commande :

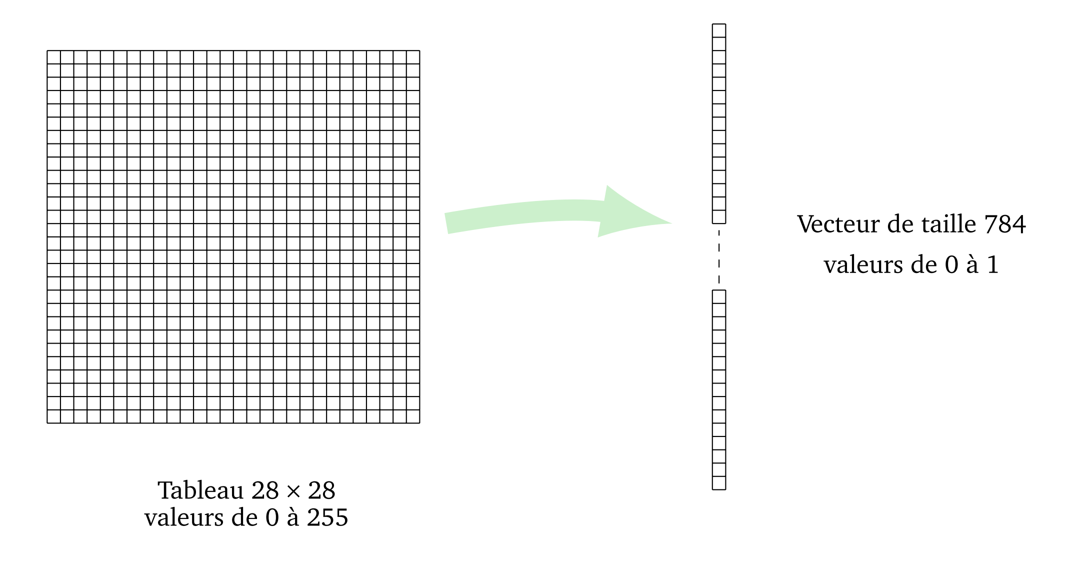
```
plt.imshow(X_train_data[i], cmap='Greys')
```

suivie de `plt.show()`.

Traitement des données

Pour une utilisation par un réseau de neurones nous devons d'abord transformer les données.

Donnée d'entrée. En entrée du réseau de neurones, nous devons avoir un vecteur. Au départ chaque image est un tableau de taille 28×28 ayant des entrées entre 0 et 255. Nous la transformons en un vecteur de taille $784 = 28^2$ et nous normalisons les données dans l'intervalle $[0, 1]$ (en divisant par 255).



Ainsi, une entrée X est un « vecteur-image », c'est-à-dire un vecteur de taille 784 représentant une image.

Donnée de sortie. Notre réseau de neurones ne va pas renvoyer le chiffre attendu, mais une liste de 10 probabilités. Ainsi chaque chiffre doit être codé par une liste de 0 et de 1.

- 0 est codé par $(1,0,0,0,0,0,0,0,0,0)$,
- 1 est codé par $(0,1,0,0,0,0,0,0,0,0)$,
- 2 est codé par $(0,0,1,0,0,0,0,0,0,0)$,
- ...
- 9 est codé par $(0,0,0,0,0,0,0,0,0,1)$.

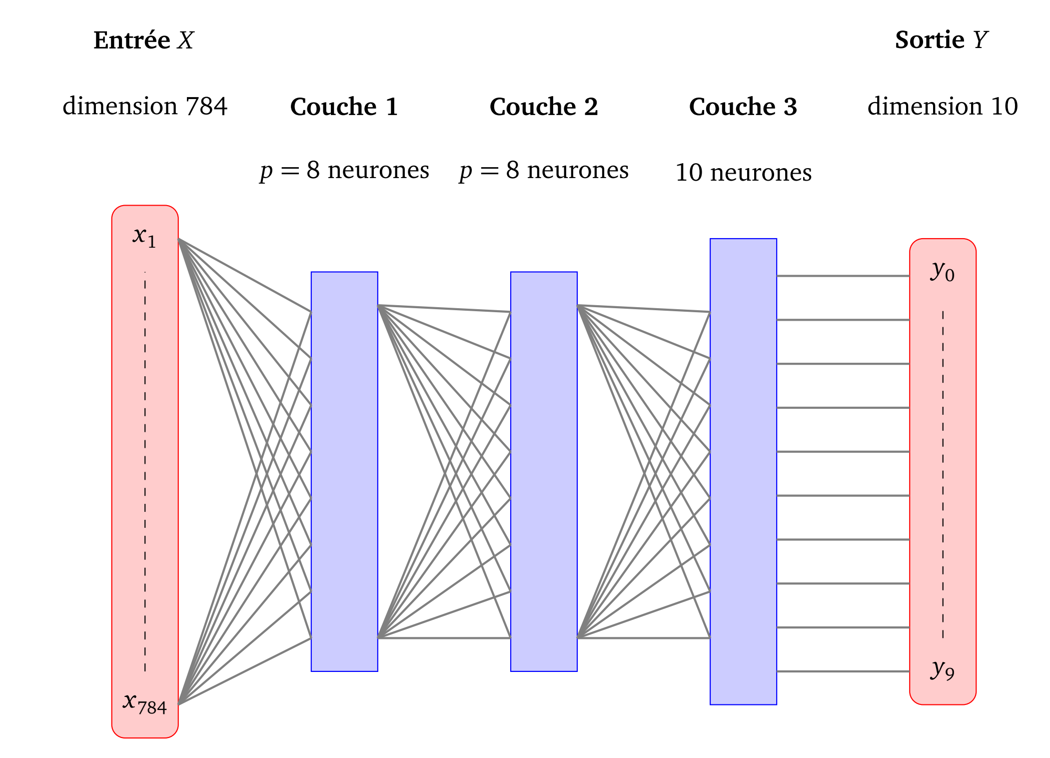
Fonction. Nous cherchons une fonction $F : \mathbb{R}^{784} \to \mathbb{R}^{10}$, qui à un vecteur-image associe une liste de probabilités, telle que $F(X_i) \simeq Y_i$ pour nos données transformées (X_i, Y_i), $i = 1, \ldots, 60\,000$.
Par exemple la fonction F, évaluée sur un vecteur-image X peut renvoyer

$$F(X) = (0.01, 0.04, 0.03, 0.01, 0.02, 0.22, 0.61, 0.02, 0.01, 0.01, 0.02).$$

Dans ce cas, le nombre le plus élevé est 0.61 au rang 6, cela signifie que notre fonction F prédit le chiffre 6 avec une probabilité de 61%, mais cela pourrait aussi être le chiffre 5 qui est prédit à 22%. Les autres chiffres sont peu probables.

Réseau

On va construire un réseau de neurones qui produira une fonction $F : \mathbb{R}^{784} \to \mathbb{R}^{10}$. L'architecture est composée de 3 couches. En entrée nous avons un vecteur de taille 784. La première et la seconde couches sont composées chacune de $p = 8$ neurones. La couche de sortie est formée de 10 neurones, un pour chacun des chiffres.



1.2. Programme

Voici le code complet du programme qui sera commenté plus loin.

```
import numpy as np
from tensorflow import keras
```

```
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

### Partie A - Les données

from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Téléchargement des données
(X_train_data,Y_train_data),(X_test_data,Y_test_data) = mnist.load_data()

N = X_train_data.shape[0]  # N = 60 000 données

# Données d'apprentissage X
X_train = np.reshape(X_train_data,(N,784))  # vecteur image
X_train = X_train/255  # normalisation

# Données d'apprentissage Y vers une liste de taille 10
Y_train = to_categorical(Y_train_data, num_classes=10)

# Données de test
X_test = np.reshape(X_test_data,(X_test_data.shape[0],784))
X_test = X_test/255
Y_test = to_categorical(Y_test_data, num_classes=10)

### Partie B - Le réseau de neurones

p = 8
modele = Sequential()

# Première couche : p neurones (entrée de dimension 784 = 28x28)
modele.add(Dense(p, input_dim=784, activation='sigmoid'))

# Deuxième couche : p neurones
modele.add(Dense(p, activation='sigmoid'))

# Couche de sortie : 10 neurones (un par chiffre)
modele.add(Dense(10, activation='softmax'))

# Choix de la méthode de descente de gradient
modele.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

print(modele.summary())

### Partie C - Calcul des poids par descente de gradient
```

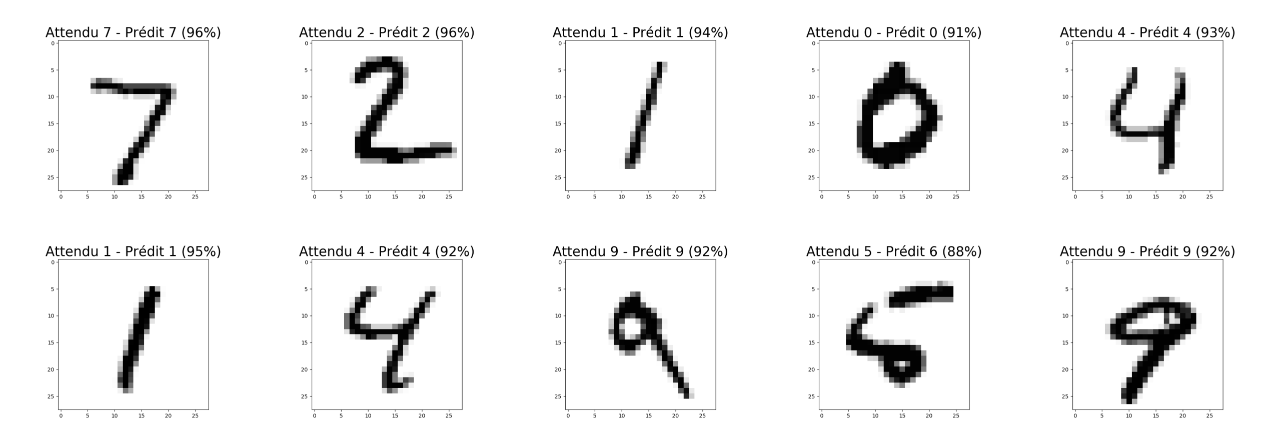
```
modele.fit(X_train, Y_train, batch_size=32, epochs=40)

### Partie D - Résultats

resultat = modele.evaluate(X_test, Y_test, verbose=0)
print('Valeur de l''erreur sur les données de test (loss):', resultat[0])
print('Précision sur les données de test (accuracy):', resultat[1])
```

1.3. Résultats

On estime la performance de notre réseau sur les données de test. Voici les premiers résultats.



Le réseau prédit correctement 9 valeurs sur 10. En fait, la fonction associée au réseau renvoie une liste de probabilités. Le chiffre prédit est celui qui a la plus forte probabilité. Par exemple pour la première image (en haut à gauche) la valeur renvoyée est :

$$Y_0 = (0.001, 0.000, 0.000, 0.008, 0.002, 0.005, 0.000, 0.965, 0.000, 0.020).$$

On en déduit la prédiction du chiffre 7 avec une forte probabilité de 96%.
L'avant-dernière image conduit à une mauvaise prédiction : la fonction prédit le chiffre 6 alors que le résultat attendu est le chiffre 5.
Les calculs de la descente de gradient sont faits avec une fonction d'erreur qu'il s'agit de minimiser. Par contre cette fonction d'erreur n'est pas pertinente pour évaluer la qualité de la modélisation. On préfère ici calculer la ***précision*** (*accuracy*) qui correspond à la proportion de chiffres détectés correctement.
Voici quelques résultats pour différentes tailles du réseau (couche 1 avec p neurones, couche 2 avec aussi p neurones, couche 3 avec 10 neurones) :

p	neurones	poids	précision
8	26	6442	90.0%
10	30	8070	91.4%
20	50	16 330	93.4%
50	110	42 310	94.4%

Sans trop d'efforts on obtient donc une précision de 95%. C'est-à-dire que 95 fois sur 100 le chiffre prédit est le chiffre correct.
Il est important de mesurer la précision sur les données de test qui sont des données qui n'ont pas été utilisées lors de l'apprentissage. Le réseau n'a donc pas appris par cœur les données d'apprentissage, mais a réussi à dégager un schéma, validé sur des données indépendantes.

1.4. Explications

Reprenons pas à pas le programme et donnons quelques explications.

Partie A - Les données

Les données d'apprentissage sont téléchargées facilement par une seule instruction. Elles regroupent $N = 60\,000$ données d'apprentissage (*train*) et 10 000 données de test. Les données sont de la forme (X_i, Y_i) où X_i est une image (un tableau 28×28 d'entiers de 0 à 255) et Y_i est le chiffre correspondant (de 0 à 9). Les données X_i sont transformées en un vecteur de taille 784 (fonction `reshape()`) et ses coefficients sont ramenés dans l'intervalle $[0, 1]$ par division par 255. Ainsi `X_train` est maintenant une liste *numpy* de 60 000 vecteurs de taille 784.
Chaque donnée Y_i est transformée en une liste de longueur 10 du type `[0,0,...,0,1,0,...,0]` avec le 1 à la place du chiffre attendu. On utilise ici la fonction `to_categorical()`.

Partie B - Le réseau de neurones

Notre réseau est composé de 3 couches. La première couche contient $p = 8$ neurones et reçoit en entrée 784 valeurs (une pour chaque pixel de l'image). La seconde couche contient aussi $p = 8$ neurones. La troisième couche contient 10 neurones (le premier pour détecter le chiffre 0, le deuxième pour le chiffre 1,...). Pour les deux premières couches la fonction d'activation est la fonction σ. Pour la couche de sortie, la fonction d'activation est la fonction *softmax* qui est adaptée au problème. Ce qui fait que la couche de sortie renvoie une liste de 10 nombres dont la somme est 1 et qui correspond à une liste de probabilités.
La fonction d'erreur (*loss*) adaptée au problème s'appelle `categorical_crossentropy`. La méthode de minimisation de l'erreur choisie est la descente de gradient stochastique.
La valeur `accuracy` du paramètre `metrics` indique que l'on souhaite en plus mesurer la précision des résultats (cela ne change rien pour la descente de gradient qui dépend uniquement de la fonction d'erreur et pas de cette précision).

Partie C - Calcul des poids par descente de gradient

La fonction `fit()` lance la descente de gradient (avec une initialisation aléatoire des poids). L'option `batch_size` détermine la taille du lot (*batch*). On indique aussi le nombre d'époques à effectuer (avec `epochs=40`, chacune des $N = 60\,000$ données sera utilisée 40 fois, mais comme chaque étape regroupe un lot de 32 données, il y $40 \times N/32$ étapes de descente de gradient).

Partie D - Résultats

On insiste sur le fait que la performance du réseau avec les poids calculés doit être mesurée sur les données de test et non sur les données d'apprentissage.
La fonction `evaluate()` renvoie la valeur de la fonction d'erreur (qui est la fonction minimisée par la descente de gradient mais qui n'a pas de signification tangible). Ici la fonction renvoie aussi la précision (car on l'avait demandée en option dans la fonction `compile()`).

Un peu plus de résultats

Pour l'instant, nous avons juste mesuré l'efficacité globale du réseau. Il est intéressant de vérifier à la main les résultats. Les instructions ci-dessous calculent les prédictions pour toutes les données de test (première ligne). Ensuite, pour une donnée particulière, on compare le chiffre prédit et le chiffre attendu.

```
# Prédiction sur les données de test
Y_predict = modele.predict(X_test)
```

```
# Un exemple
i = 8  # numéro de l'image

chiffre_predit = np.argmax(Y_predict[i]) # prédiction par le réseau

print("Sortie réseau", Y_predict[i])
print("Chiffre attendu :", Y_test_data[i])
print("Chiffre prédit :", chiffre_predit)

plt.imshow(X_test_data[i], cmap='Greys')
plt.show()
```

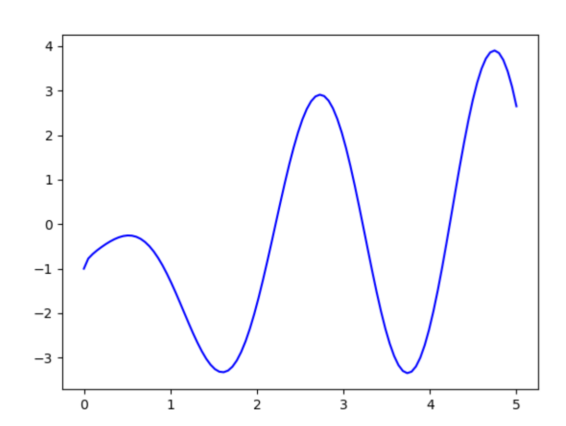
On rappelle que `Y_predict[i]` est une liste de 10 nombres correspondant à la probabilité de chaque chiffre. Le chiffre prédit s'obtient en prenant le rang de la probabilité maximale, c'est exactement ce que fait la fonction `argmax` de *numpy*.

2. Fonction d'une variable

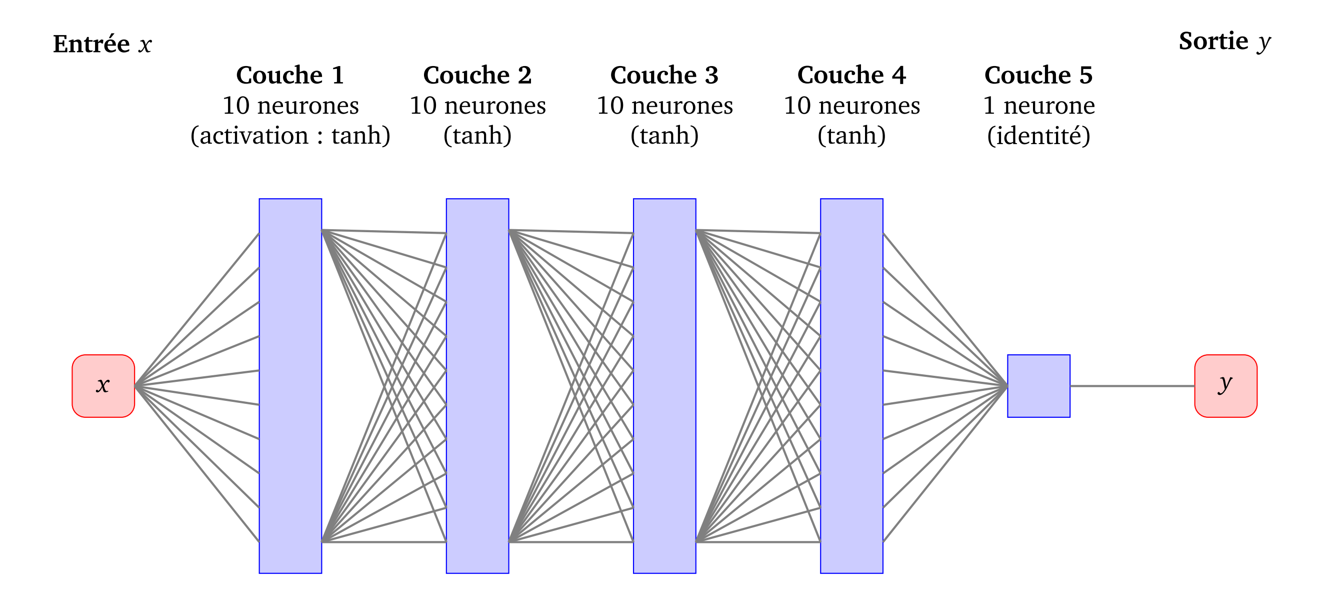
2.1. Données

Le but est de construire un réseau et de calculer ses poids afin d'approcher la fonction :

$$f(x) = \cos(2x) + x\sin(3x) + \sqrt{x} - 2 \qquad \text{avec } \; x \in [0,5].$$



On divise l'intervalle de départ $[0,5]$ pour obtenir $N = 100$ abscisses x_i qui forment la première partie des données d'apprentissage `X_train`. On calcule ensuite les $y_i = f(x_i)$, ce qui donne 100 ordonnées qui forment l'autre partie des données d'apprentissage `Y_train`.
On propose un réseau avec 4 couches de 10 neurones, tous de fonction d'activation la fonction tangente hyperbolique, et d'une couche de sortie formée d'un seul neurone de fonction d'activation l'identité. L'entrée et la sortie sont de dimension 1. La fonction associée au réseau est donc $F : \mathbb{R} \to \mathbb{R}$. On souhaite calculer les poids du réseau de sorte que $F(x) \simeq f(x)$, pour tout $x \in [0,5]$.



2.2. Programme

```
# Partie A. Données

# Fonction à approcher
def f(x):
    return np.cos(2*x) + x*np.sin(3*x) + x**0.5 - 2

a, b = 0, 5                 # intervalle [a,b]
N = 100                     # taille des données
X = np.linspace(a, b, N)    # abscisses
Y = f(X)                    # ordonnées
X_train = X.reshape(-1,1)
Y_train = Y.reshape(-1,1)

# Partie B. Réseau

modele = Sequential()

p = 10
modele.add(Dense(p, input_dim=1, activation='tanh'))
modele.add(Dense(p, activation='tanh'))
modele.add(Dense(p, activation='tanh'))
modele.add(Dense(p, activation='tanh'))
modele.add(Dense(1, activation='linear'))

# Méthode de gradient : descente de gradient classique améliorée
mysgd = optimizers.SGD(lr=0.001, decay=1e-7, momentum=0.9, nesterov=True)
modele.compile(loss='mean_squared_error', optimizer=mysgd)
print(modele.summary())

# Partie C. Apprentissage

history = modele.fit(X_train, Y_train, epochs=4000, batch_size=N)
```

```
# Partie D. Visualisation

# Affichage de la fonction et de son approximation
Y_predict = modele.predict(X_train)
plt.plot(X_train, Y_train, color='blue')
plt.plot(X_train, Y_predict,  color='red')
plt.show()

# Affichage de l'erreur au fil des époques
plt.plot(history.history['loss'])
plt.show()
```

2.3. Explications

On a personnalisé la méthode de descente de gradient par la commande

```
mysgd = optimizers.SGD(lr=0.001, decay=1e-7, momentum=0.9, nesterov=True)
```

et on demande à utiliser ces paramètres dans la ligne suivante :

```
modele.compile(loss='mean_squared_error', optimizer=mysgd)
```

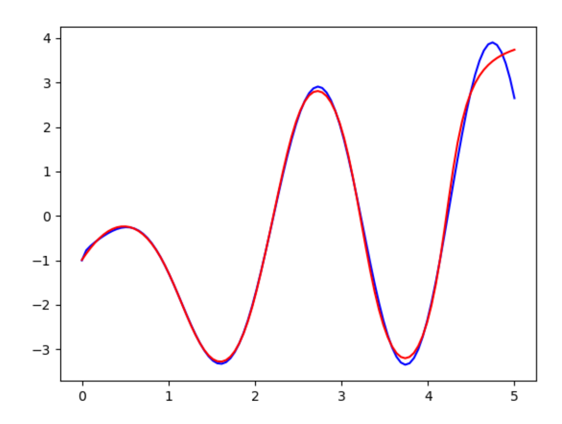
La fonction d'erreur est ici l'erreur quadratique moyenne. Notre méthode de gradient débute avec un taux d'apprentissage $\delta = 0.001$ (variable `lr` pour *learning rate*). Ce taux d'apprentissage va diminuer à chaque étape de la descente de gradient selon un paramètre $\alpha = 10^{-7}$ (option `decay`, voir la section 2 du chapitre « Descente de gradient »). Enfin, on utilise un moment (*momentum*) et l'accélération de Nesterov (voir la section 4 du chapitre « Descente de gradient »).
Finalement, on lance le calcul des poids :

```
history = modele.fit(X_train, Y_train, epochs=4000, batch_size=N)
```

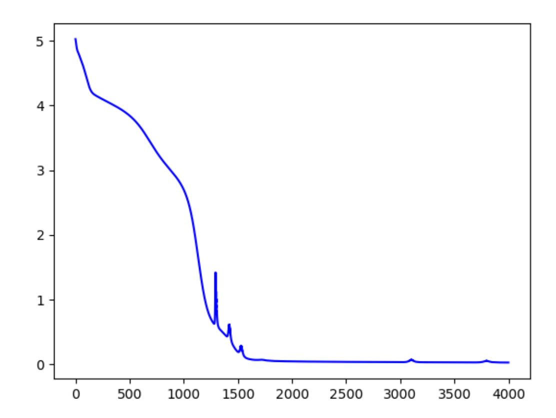
Ici on effectue 4000 époques. La taille de l'échantillon est égale à la taille des données N, c'est une descente de gradient classique, il y a donc aussi 4000 étapes dans la descente de gradient. La variable `history` contient l'historique des valeurs renvoyées par la fonction `fit()` et permet par exemple d'afficher la valeur de l'erreur au fil des époques.

2.4. Résultats

Le réseau fournit donc une fonction $F : \mathbb{R} \to \mathbb{R}$ qui approche correctement la fonction f sur l'intervalle $[0, 5]$.

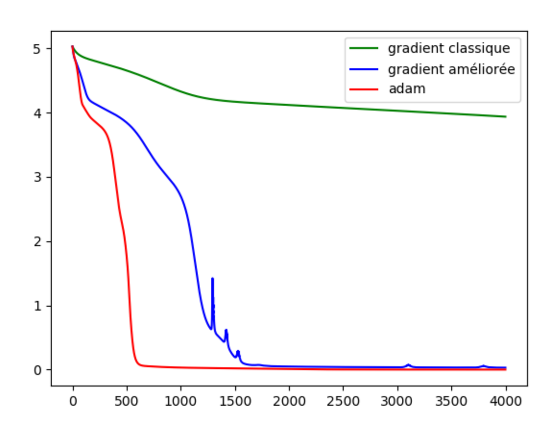


Voici le comportement de l'erreur au fil des époques.



L'erreur n'est pas partout une fonction décroissante, c'est le cas lorsque le taux d'apprentissage est trop grand. D'où l'intérêt de faire diminuer ce taux d'apprentissage au fil des époques à l'aide du paramètre α.
Pour terminer, comparons l'évolution des erreurs selon trois méthodes différentes de descente de gradient :

- la descente de gradient classique avec $\delta = 0.001$;
- la descente de gradient classique améliorée (comme dans le programme ci-dessus) avec $\delta = 0.001$ au départ, mais qui décroît ensuite selon $\alpha = 10^{-7}$, avec un moment et l'accélération de Nesterov ;
- la descente de gradient « *adam* », qui est une méthode récente et performante, mais plus compliquée que les précédentes.



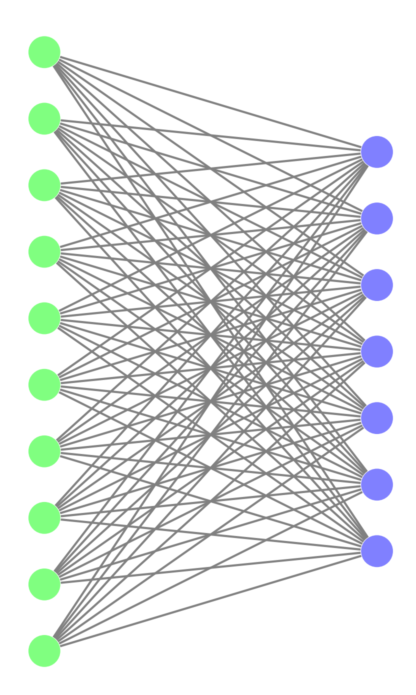
On constate des différences importantes de performance. En particulier, pour la méthode classique, il faudrait poursuivre encore longtemps les itérations pour obtenir une erreur raisonnable.

3. Quelques fonctionnalités de *tensorflow*/*keras*

On revient un peu plus en détails sur l'utilisation de *tensorflow*/*keras*.

3.1. Architecture du réseau

Pour l'instant nous ne travaillons qu'avec un seul type d'architecture de réseaux : des réseaux formés d'une succession de couches de neurones, avec entre deux couches toutes les connexions possibles. On parle de réseaux complètement connectés.



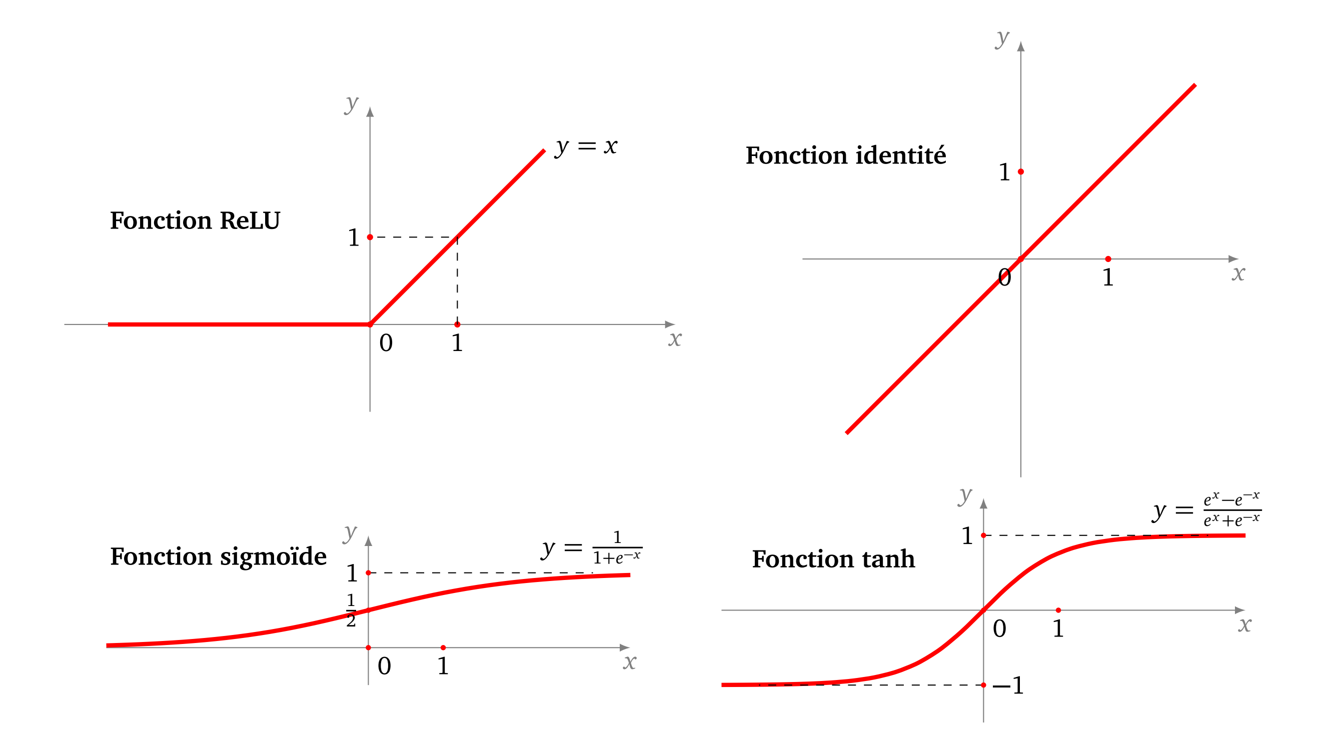
Avec *tensorflow*/*keras*, il suffit de déclarer le modèle du réseau en ajoutant une à une les couches.

```
modele = Sequential()
modele.add(Dense(N1, input_dim=n, activation='...'))   # première couche
modele.add(Dense(N2, activation='...'))                # deuxième couche
...
modele.add(Dense(p, activation='...'))                 # dernière couche
```

La fonction `add()` prend en entrée le nombre N_i de neurones sur cette couche, ainsi que la fonction d'activation des neurones de cette couche.
Pour la première couche (appelée couche d'entrée), il faut également indiquer la dimension des données en entrée par l'option `input_dim=n` où n est la dimension de chaque donnée d'entrée. Le nombre de neurones p de la dernière couche (appelée couche de sortie) détermine la dimension de chaque donnée de sortie. Ainsi un réseau ayant des entrées de dimension n, et p neurones en sortie, détermine une fonction $F : \mathbb{R}^n \to \mathbb{R}^p$.

3.2. Fonctions d'activation

- **La fonction ReLU** (pour *Rectified Linear Unit*) s'utilise par l'option `activation='relu'`. Elle est définie par
$$\begin{cases} f(x) = 0 & \text{si } x < 0 \\ f(x) = x & \text{si } x \geqslant 0 \end{cases}$$
- **Fonction identité** (option `activation='linear'`) définie par $f(x) = x$. Elle est utile par exemple pour un neurone de sortie.
- **La fonction sigmoïde** (option `activation='sigmoid'`) est définie par :
$$\sigma(x) = \frac{1}{1+e^{-x}}.$$
Elle prend ses valeurs dans $[0,1]$. C'est une version continue de la fonction marche de Heaviside.
- **La fonction tangente hyperbolique** (option `activation='tanh'`) est définie par :
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$
C'est une variante de la fonction sigmoïde qui prend ses valeurs dans $[-1,1]$.



3.3. Fonctions d'erreur

Pour calculer de bons poids par la descente de gradient, il faut calculer l'erreur entre la sortie attendue et la sortie produite par le réseau. Il faut donc spécifier la méthode utilisée pour calculer cette erreur lorsque l'on finalise le modèle par la commande `compile(loss='...')`. Plus de détails seront donnés dans le chapitre « Probabilités ».

Si l'on possède N données d'apprentissage, notons y_i les sorties attendues et $\widetilde{y}_i$ les sorties produites (calculées par le réseau $\widetilde{y}_i = F(x_i)$).

- **Erreur quadratique moyenne** (option `loss='mean_squared_error'`). C'est la moyenne des distances au carré :
$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - \widetilde{y}_i)^2.$$
Si l'erreur vaut 0 c'est que toutes les valeurs prédites sont exactement les valeurs attendues.
- **Erreur absolue moyenne** (option `loss='mean_absolute_error'`). C'est la moyenne des distances :
$$E = \frac{1}{N}\sum_{i=1}^{N}|y_i - \widetilde{y}_i|.$$
Intuitivement, cette définition est plus naturelle car elle mesure une distance (et non une distance au carré), mais l'usage de la valeur absolue la rend moins aisée que la précédente.
- **Erreur logarithmique moyenne** (option `loss='mean_squared_logarithmic_error'`) qui est :
$$E = \frac{1}{N}\sum_{i=1}^{N}\big(\ln(y_i + 1) - \ln(\widetilde{y}_i + 1)\big)^2.$$
L'usage du logarithme est adapté à des données de différentes tailles, car elle prend en compte de façon identique une petite erreur sur une petite valeur et une grande erreur sur une grande valeur.
- **Entropie croisée binaire** (option `loss='binary_crossentropy'`). Elle est adaptée lorsque le problème a pour sortie attendue $y_i = 0$ ou $y_i = 1$ et que la sortie produite est une probabilité $\widetilde{y}_i$ avec $0 \leqslant \widetilde{y}_i \leqslant 1$.
- **Entropie croisée pour catégories** (option `loss='categorical_crossentropy'`). Elle est adaptée lorsque le problème a pour sortie attendue un vecteur du type $y_i = (0, 0, \ldots, 1, \ldots, 0)$ (avec un seul 1 à

la place qui désigne le rang de la catégorie attendue) et la sortie produite est une liste de probabilités $\widetilde{y}_i = (p_1, p_2, \ldots)$ (voir l'exemple de la reconnaissance de chiffres).

3.4. Descente de gradient

Il faut préciser quelle méthode va être utilisée pour minimiser la fonction d'erreur. Cela se fait lors de la finalisation du modèle par la commande `compile(optimizer=...)`. Par contre, la taille de l'échantillon (*batch*) sera précisée plus tard, lors de l'utilisation de la commande `fit()`.

- **Descente de gradient stochastique classique** (option `optimizer='sgd'`). C'est la méthode classique telle qu'elle a été expliquée dans le chapitre « Descente de gradient ». La taille de l'échantillon permet de faire une descente de gradient stochastique pure (`batch_size=1`), une descente de gradient sur la totalité des N données (`batch_size=N`) ou toute solution intermédiaire (par exemple `batch_size=32`).
- **Descente de gradient personnalisée améliorée.** Par exemple
```
mysgd = optimizers.SGD(lr=0.001, decay=1e-7, momentum=0.9, nesterov=True)
```
  puis un appel par l'option `optimizer=mysgd`.
  Les paramètres sont :
  — le taux d'apprentissage δ (variable `lr`),
  — le taux α de décroissance de δ (option `decay`),
  — le moment (`momentum`),
  — et l'utilisation ou pas de l'accélération de Nesterov.
  Voir de nouveau le chapitre « Descente de gradient » pour les détails.

Les méthodes de descente sont l'objet de recherches intenses et ont fait d'énormes progrès. Les nouvelles méthodes sont beaucoup plus performantes et accélèrent grandement les calculs. Mais ces dernières sont trop compliquées et au-delà des objectifs de ce livre, nous n'en donnerons pas d'explications. En voici quelques-unes à tester, parmi celles-ci la méthode « adam » est l'une des meilleures :

- `'adagrad'`
- `'adadelta'`
- `'rmsprop'`
- `'adam'`

3.5. Mise en place du modèle

On finalise le modèle par la fonction `compile()`.
```
modele.compile(loss=..., optimizer=...)
```
qui précise le choix de la fonction d'erreur et la méthode de minimisation. Le réseau est alors initialisé avec des poids aléatoires.
Une fois mis en place, on obtient des informations sur le modèle par la commande :
```
print(modele.summary())
```
qui fournit un résumé chiffré du réseau avec le nombre de couches, le nombre de neurones par couche et le nombre total de poids (coefficients et biais) à calculer.
On peut vouloir mesurer d'autres choses que la fonction d'erreur. Par exemple l'option de la fonction `compile()` :
```
metrics=['accuracy']
```
va en plus mémoriser la précision du modèle (sous la forme d'un pourcentage de réussite). Par exemple, si on doit classer des images en deux catégories (chat/0 et chien/1) alors la fonction d'erreur est un outil indispensable pour notre problème, mais le résultat est mesuré concrètement par le pourcentage d'images correctement identifiées.

Note : pourquoi ne pas prendre la précision comme fonction d'erreur puisque c'est ce qui nous intéresse ? Tout simplement parce que ce n'est pas un fonction différentiable, il n'y a donc pas de méthode de gradient pour la minimiser.

3.6. Calcul des poids

On lance le calcul des poids par la fonction `fit()` :

```
modele.fit(X_train, Y_train, batch_size=32, epochs=100)
```

- Les calculs sont effectués selon la méthode de descente de gradient choisie auparavant.
- Les données d'apprentissage utilisées sont `X_train` (valeurs de départ) et `Y_train` (valeurs d'arrivée attendues).
- L'option `batch_size` précise la taille de l'échantillon :
  — une descente de gradient stochastique pure : `batch_size=1`,
  — une descente de gradient sur la totalité des N données : `batch_size=N`,
  — ou toute valeur intermédiaire : par défaut `batch_size=32`.
- L'option `epochs` détermine le nombre d'étapes dans la méthode de gradient. Si K est la taille d'un lot (valeur de `batch_size`) et N la taille des données alors le nombre d'étapes par époque est N/K.
- Il existe une option `verbose` qui permet d'afficher plus ou moins de détails à chaque époque (0 : rien, 1 : barre de progression, 2 : numéro de l'époque).

3.7. Validation et prédiction

Une fois les poids calculés, on peut prédire des résultats, c'est-à-dire calculer $F(X)$ pour n'importe quel $X \in \mathbb{R}^n$ et pas seulement pour les X_i des données d'apprentissage. Cela se fait par la fonction `predict()` :

```
Y_predict = modele.predict(X)
```

Le calcul des poids par la descente de gradient a pour but de minimiser la fonction d'erreur en utilisant les données d'apprentissage. Mais la pertinence du réseau se mesure sur des données de test qui doivent être indépendantes des données d'apprentissage. La commande

```
resultat = modele.evaluate(X_test, Y_test)
```

renvoie la fonction d'erreur calculée sur les données de test (et éventuellement les valeurs demandées par l'option `metrics`). Si tout va bien, l'erreur sur les données de test devrait être proche de l'erreur sur les données d'apprentissage.

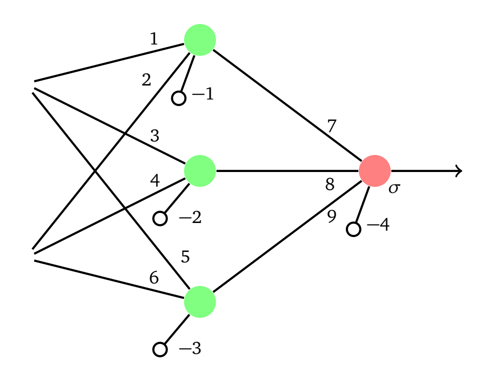
3.8. Poids

Poids.

La commande `modele.get_weights()` renvoie tous les poids du réseau. Les poids sont renvoyés sous la forme d'une liste [coefficients de la couche 1, biais de la couche 1, coefficients de la couche 2, ...]. On peut aussi travailler couche par couche (voir « Python : tensorflow avec keras - partie 1 »).

- Les biais sont donnés sous la forme d'un vecteur (un nombre pour chaque neurone).
- Les coefficients sont donnés sous la forme d'un tableau à deux dimensions, dans un ordre contre-intuitif : d'abord les poids de la première entrée pour chaque neurone, puis les poids de la seconde entrée pour chaque neurone, etc.

```
poids = [ np.array([[1.,3.,5.], [2.,4.,6.]]),  # Coeff. couche 1
          np.array([-1.,-2.,-3.]),              # Biais couche 1
          np.array([[7.], [8.], [9.]]),         # Coeff. couche 2
          np.array([-4.]) ]                     # Biais couche 2
```



La commande `modele.set_weights(poids)` permet de définir les poids à la valeur voulue.

Calculs pas à pas. La méthode `fit()` effectue un calcul avec un nombre d'époques donné. On peut travailler étape par étape avec une commande du type :

```
loss = modele.train_on_batch(X_train, Y_train)
```

qui correspond à une étape de descente de gradient.

Gradient. Il n'est pas facile de récupérer la valeur du gradient avec *tensorflow/keras*. Le module `keras_facile` propose une fonction `get_weights_grad()` qui renvoie les poids du gradient :

```
gradient = get_weights_grad(modele, X_train, Y_train)
```

Les poids du gradient ont la même structure que les poids du réseau.
On peut ainsi programmer facilement à la main sa propre descente de gradient par la commande :

```
poids_apres = [poids_avant[i] - delta*gradient[i] for i in
                      range(len(poids_avant))]
```

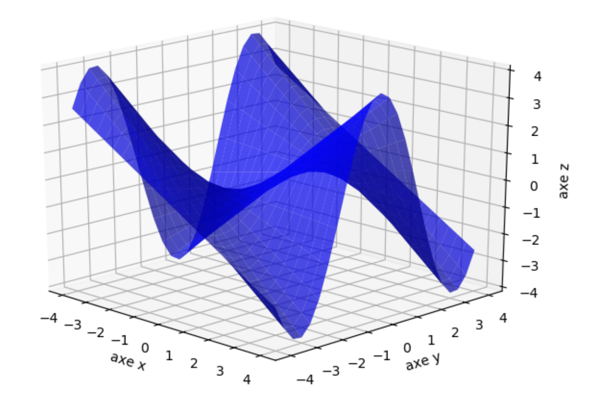
où `delta` est le taux d'apprentissage (*learning rate*, $\delta = 0.1$ par exemple). Voir le fichier `poids_gradient.py` pour un exemple.

4. Fonction de deux variables

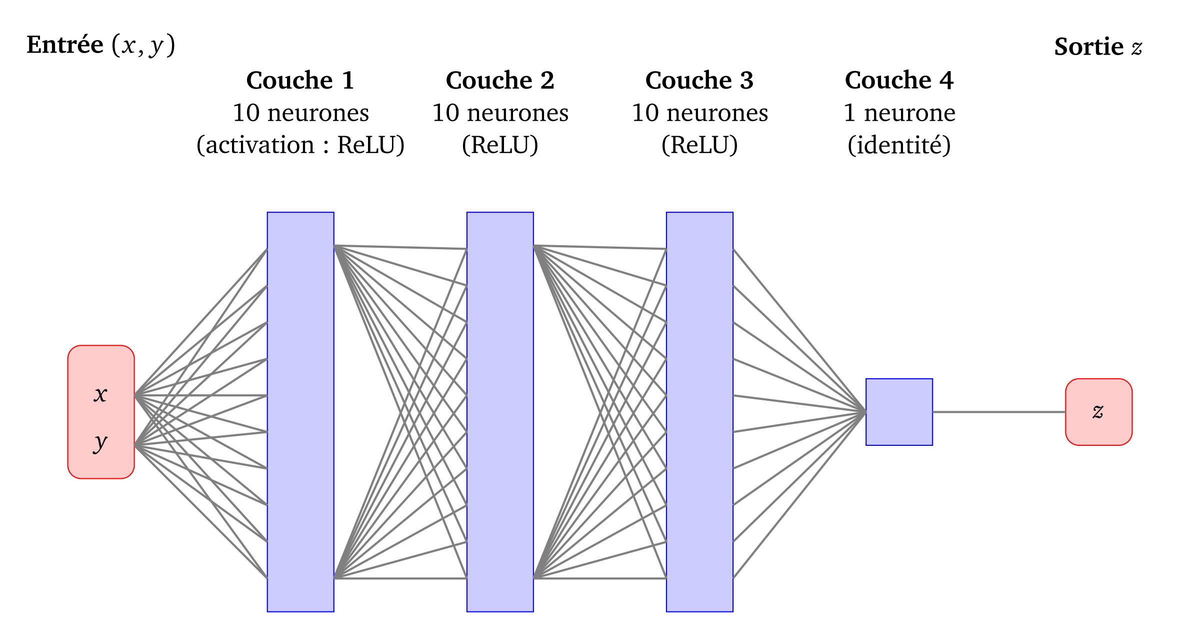
4.1. Données

Nous souhaitons construire un réseau pour approcher la fonction de deux variables :

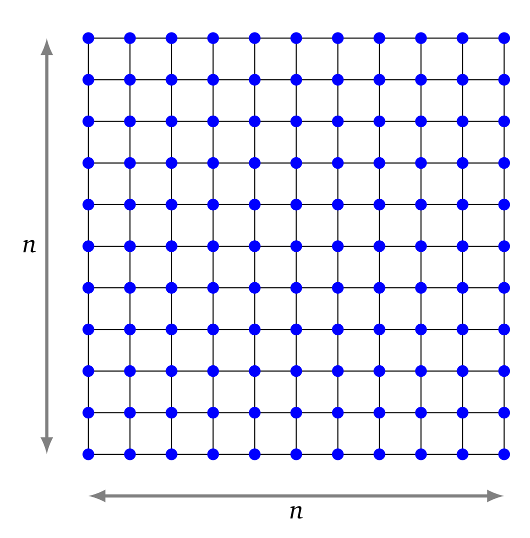
$$f(x,y) = x\cos(y) \text{ avec } \ x \in [-4,4] \text{ et } y \in [-4,4].$$



On construit un réseau de plusieurs couches, avec deux entrées et une sortie, auquel est donc associée une fonction $F : \mathbb{R}^2 \to \mathbb{R}$. On souhaite que $F(x,y) \simeq f(x,y)$ pour $(x,y) \in [-4,4]^2$. La fonction d'activation est la fonction « ReLU » (sauf pour le neurone de la couche de sortie qui a pour fonction d'activation l'identité).



Les données d'apprentissage sont du type (x_i, y_i, z_i) avec d'une part des (x_i, y_i) sur une grille du carré $[-4, 4]^2$ et $z_i = f(x_i, y_i)$.



4.2. Programme

```
# Partie A. Données

# Fonction à approcher
def f(x,y):
    return x*np.cos(y)

n = 25  # pour le nb de points dans la grille
xmin, xmax, ymin, ymax = -4.0, 4.0, -4.0, 4.0

VX = np.linspace(xmin, xmax, n)
VY = np.linspace(ymin, ymax, n)
X, Y = np.meshgrid(VX, VY)
Z = f(X, Y)

entree = np.append(X.reshape(-1,1), Y.reshape(-1,1), axis=1)
```

```
sortie = Z.reshape(-1, 1)

# Partie B. Réseau

modele = Sequential()
p = 10
modele.add(Dense(p, input_dim=2, activation='relu'))
modele.add(Dense(p, activation='relu'))
modele.add(Dense(p, activation='relu'))
modele.add(Dense(1, activation='linear'))

# Méthode de gradient : descente de gradient classique
mysgd = optimizers.SGD(lr=0.01)
modele.compile(loss='mean_squared_error', optimizer=mysgd)
print(modele.summary())

# Partie C. Apprentissage

# Apprentissage époque par époque à la main
for k in range(1000):
    loss = modele.train_on_batch(entree, sortie)
    print('Erreur :',loss)

# Partie D. Visualisation

sortie_produite = modele.predict(entree)
ZZ = sortie_produite.reshape(Z.shape)  # sortie aux bonnes dimensions

# Affichage
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

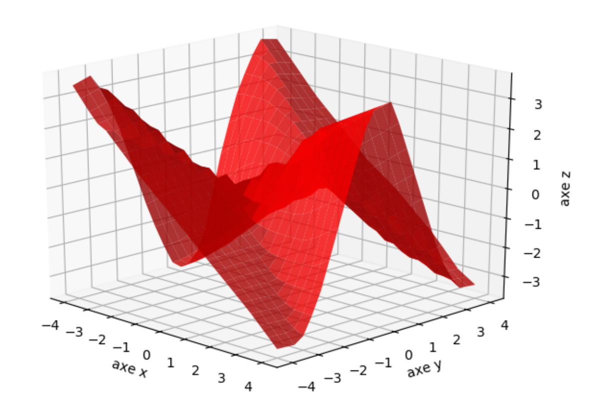
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, color='blue', alpha=0.7)
ax.plot_surface(X, Y, ZZ, color='red', alpha=0.7)
plt.show()
```

4.3. Explications

On a ici utilisé la fonctionnalité `train_on_batch()` qui permet d'effectuer une étape de gradient en tenant compte de la totalité des données (cela correspond donc à une époque).

4.4. Résultats

La fonction prédite est proche de la fonction voulue.



5. Reconnaissance de texte

5.1. Données

Le but de cet exemple est de décider si une critique de film est positive ou négative. Voici un exemple de critique (la critique numéro 123) de la base que nous utiliserons :

> *beautiful and touching movie rich colors great settings good acting and one of the most charming movies i have seen in a while i never saw such an interesting setting when i was in china my wife liked it so much she asked me to log on and rate it so other would enjoy too*

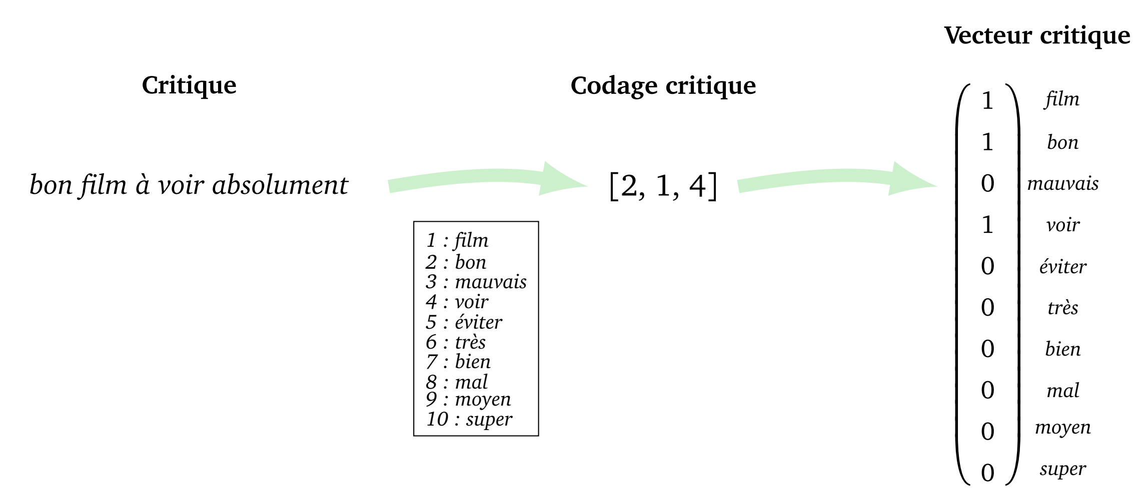
C'est un critique positive !
La base IMDB fournit 25 000 critiques d'apprentissage, chacune étant déjà catégorisée positive (valeur 1) ou négative (valeur 0). Le réseau de neurones pour ce problème va fournir une fonction $F : \mathbb{R}^n \to \mathbb{R}$. Si $F(X) \simeq 1$ alors la critique est positive, si $F(X) \simeq 0$ elle est négative.
La première question est : comment transformer le texte d'une critique en un vecteur X de $\mathbb{R}^n$? Nous allons l'expliquer sur un exemple simpliste.
Voici trois (fausses) critiques :

« *bon film à voir absolument* »
« *début moyen mais après c'est très mauvais* »
« *film drôle et émouvant on passe un bon moment* »

- Tout d'abord, parmi toutes les critiques, on ne retient que les 10 mots les plus fréquents. Imaginons que ce sont les mots :

  film, bon, mauvais, voir, éviter, très, bien, mal, moyen, super

  En réalité on retiendra les 1000 (voire 10 000) mots les plus fréquents.
- On code une phrase comme une liste d'indices de mots. Le mot « *film* » est remplacé par 1, le mot « *bon* » par 2,... jusqu'à « *super* ». Les autres mots ne sont pas pris en compte. Par exemple la critique « *bon film à voir absolument* » devient la liste d'indices $[2, 1, 4]$. Les deux autres critiques deviennent $[9, 3]$ et $[1, 2]$.
- On transforme ensuite chaque liste en un vecteur de taille fixe $n = 10$ de coordonnées 0 ou 1. On place un 1 en position i si le mot numéro i apparaît dans la critique. Ainsi la phrase « *bon film à voir absolument* », codée en $[2, 1, 4]$ devient le vecteur $X = (1, 1, 0, 1, 0, 0, 0, 0, 0, 0) \in \mathbb{R}^{10}$. La phrase codée $[9, 3]$ donne le vecteur $X' = (0, 0, 1, 0, 0, 0, 0, 0, 1, 0)$.



- Ainsi chaque critique est maintenant une entrée $X_i \in \mathbb{R}^n$ (avec $n = 10$ dans la version simple ci-dessus, mais en réalité ce sera plutôt $n = 1000$). Lors de l'apprentissage nous connaissons aussi la sortie attendue Y_i (0 ou 1). On peut alors construire un réseau à n entrées et une seule sortie qui fournit une fonction $F : \mathbb{R}^n \to \mathbb{R}$ avec pour objectif d'avoir $F(X_i) \simeq Y_i$ sur les données d'apprentissage.

Noter qu'avec ce codage on perd pas mal d'information par rapport à la critique initiale : il manque des mots, l'ordre des mots n'est pas respecté, les mots répétés ne sont pas pris en compte. Mais aussi, « *bon acteur mais mauvais film* » et « *bon film mais mauvais acteur* » sont codées par le même vecteur. Malgré cela, on obtient tout de même de très bonnes prédictions !

5.2. Programme

```
# Partie A. Données

from tensorflow.keras.datasets import imdb

# On ne garde que les n=1000 mots les plus fréquents
nb_mots_total = 1000
(X_train_data, Y_train), (X_test_data, Y_test) =
                        imdb.load_data(num_words = nb_mots_total)

# Partie A bis. Afficher d'un texte

# Afficher une critique et sa note
def affiche_texte(num):
    index_mots = imdb.get_word_index()
    index_mots_inverse = dict([(value, key)
                               for (key, value) in index_mots.items()])
    critique_mots = ' '.join([index_mots_inverse.get(i - 3, '??')
                              for i in X_train_data[num]])
    print("Critique :\n", critique_mots)
    print("Note 0 (négatif) ou 1 (positif) ? :", Y_train[num])
    print("Critique (sous forme brute) :\n", X_train_data[num])
    return

affiche_texte(123)   # affichage de la critique numéro 123
```

```
# Partie A ter. Données sous forme de vecteurs

def vectorisation_critiques(X_data):
    vecteurs = np.zeros((len(X_data), nb_mots_total))
    for i in range(len(X_data)):
        for c in X_data[i]:
            vecteurs[i,c] = 1.0
    return vecteurs

X_train = vectorisation_critiques(X_train_data)
X_test = vectorisation_critiques(X_test_data)

# Partie B. Réseau

modele = Sequential()
p = 5
modele.add(Dense(p, input_dim=nb_mots_total, activation='relu'))
modele.add(Dense(p, activation='relu'))
modele.add(Dense(p, activation='relu'))
modele.add(Dense(1, activation='sigmoid'))
modele.compile(loss='binary_crossentropy',
               optimizer='sgd',
               metrics=['accuracy'])

# Partie C. Apprentissage

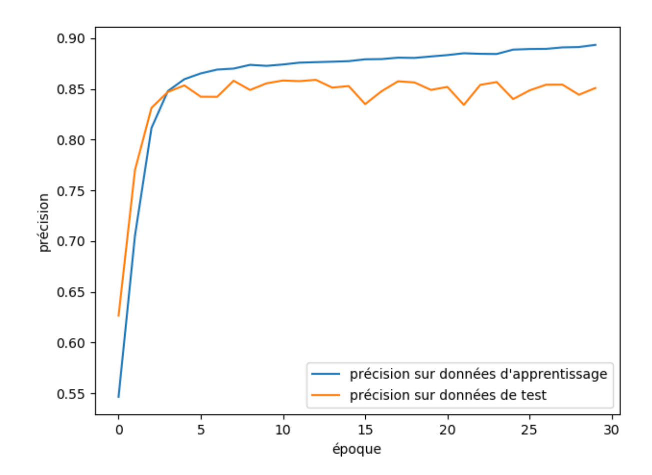
modele.fit(X_train, Y_train, epochs=10, batch_size=32)


# Partie D. Résultats

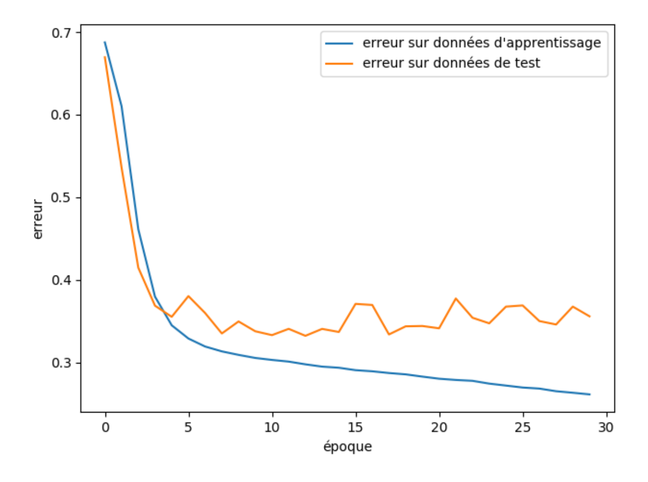
Y_predict = modele.predict(X_test)
```

5.3. Résultats

Avec un réseau de seulement 16 neurones ($5+5+5+1$), la descente de gradient stochastique classique et 10 époques, nous obtenons une précision entre 80% et 85% sur les données de tests.
Est-ce qu'on peut faire mieux en augmentant le nombre d'étapes ? Sur le diagramme ci-dessous nous comparons, au fil des époques, la précision obtenue sur les données d'apprentissage avec la précision obtenue sur les données de test.



La précision calculée sur les données d'apprentissage augmente au fil des époques. Avec un peu plus d'effort, on pourrait même obtenir un précision de 99.9%. Par contre, la précision sur les données de test augmente au tout début, puis fluctue autour de 85%.
Que se passe-t-il ? Nous sommes dans une situation de sur-apprentissage ! En augmentant le nombre d'époques, le réseau finit par apprendre par cœur les données d'apprentissage. Par contre, le réseau n'améliore pas les prévisions sur les données de test et n'est donc pas plus performant.
On retrouve ce phénomène sur l'erreur au fil des époques. Elle décroît continuellement vers 0 sur les données d'apprentissage, alors que sur les données de test, l'erreur ne diminue plus après 5 époques.



6. Reconnaissance d'images

On termine par un constat d'échec (provisoire), nos réseaux de neurones atteignent leurs limites lorsque le problème posé se complique. Nous souhaitons reconnaître des petites images et les classer selon 10 catégories.

6.1. Données

La base CIFAR-10 contient 60 000 petites images de 10 types différents.



Plus en détails :

- Il y a 50 000 images pour l'apprentissage et 10 000 pour les tests.
- Chaque image est de taille 32×32 pixels en couleur. Un pixel couleur est codé par trois entiers (r, g, b) compris entre 0 et 255. Une image est donc composée de $32 \times 32 \times 3$ nombres.
- Chaque image appartient à une des dix catégories suivantes : avion, auto, oiseau, chat, biche, chien, grenouille, cheval, bateau et camion.

6.2. Programme

```
# Partie A. Données

from tensorflow.keras.datasets import cifar10

(X_train_data,Y_train_data),(X_test_data,Y_test_data)=cifar10.load_data()

num_classes = 10
labels =  ['airplane','automobile','bird','cat','deer',
           'dog','frog','horse','ship','truck']

Y_train = keras.utils.to_categorical(Y_train_data, num_classes)
X_train = X_train_data.reshape(50000,32*32*3)
X_train = X_train.astype('float32')
X_train = X_train/255

# Partie B. Réseau

modele = Sequential()

p = 30
modele.add(Dense(p, input_dim=32*32*3, activation='sigmoid'))
modele.add(Dense(p, activation='sigmoid'))
modele.add(Dense(p, activation='sigmoid'))
```

```
modele.add(Dense(p, activation='sigmoid'))
modele.add(Dense(10, activation='softmax'))

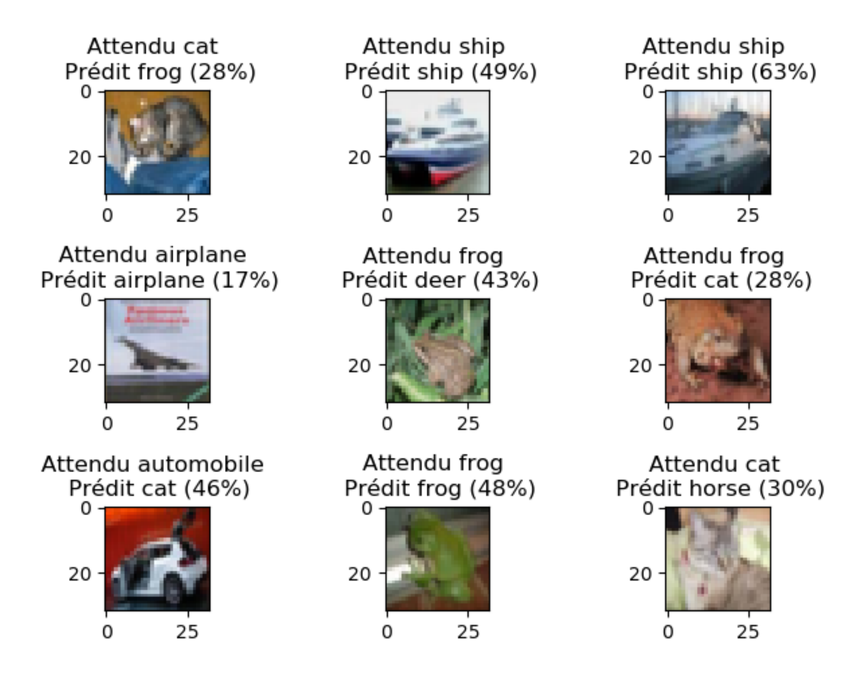
modele.compile(loss='categorical_crossentropy',
               optimizer='adam', metrics=['accuracy'])

# Partie C. Apprentissage

modele.fit(X_train, Y_train, epochs=10, batch_size=32)
```

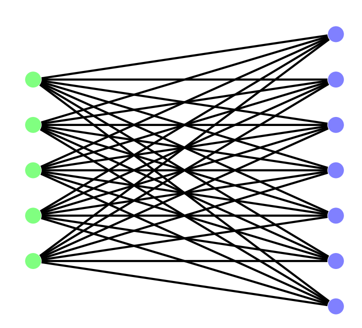
6.3. Résultats

Dans le programme ci-dessus avec $p = 30$, il y a 100 000 poids à calculer. Avec beaucoup de calculs (par exemple avec 50 époques et la descente « adam »), on obtient moins de 50% de précision. Ce qui fait que le sujet d'une image sur deux est mal prédit.



Il faut donc une nouvelle approche pour reconnaître correctement ces images !

Deuxième partie



algèbre – convolution

Convolution : une dimension

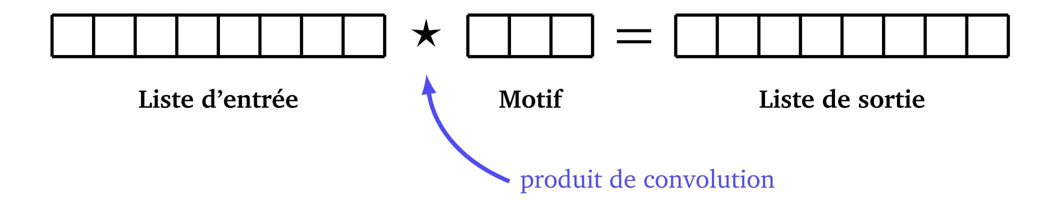
Chapitre 11

Vidéo ■ partie 11. Convolution : une dimension

Ce chapitre permet de comprendre la convolution dans le cas le plus simple d'un tableau à une seule dimension.

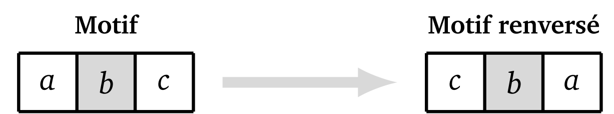
1. Idée

La convolution est une opération qui à partir d'un tableau de nombres et d'un motif produit un nouveau tableau de nombres.

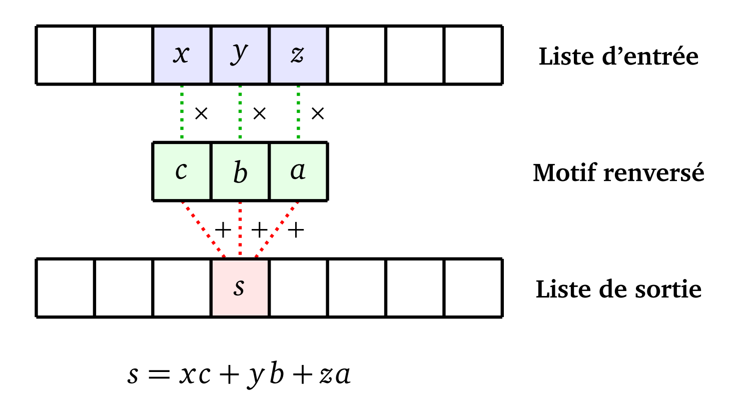


Le calcul de la liste de sortie se fait par des opérations très simples.

- On commence par renverser le motif.

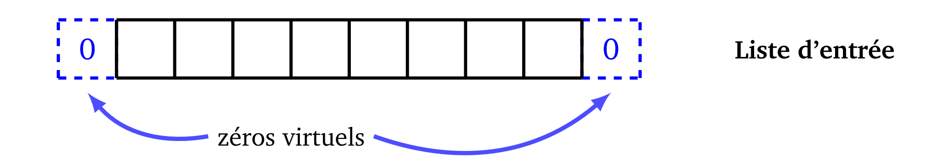


- On calcule la liste de sortie terme par terme :
  — on centre le motif renversé sous la liste d'entrée, à la position à calculer,
  — on multiplie terme à terme les éléments de la liste d'entrée et ceux du motif,
  — la somme de tous ces produits est le terme de la liste de sortie.



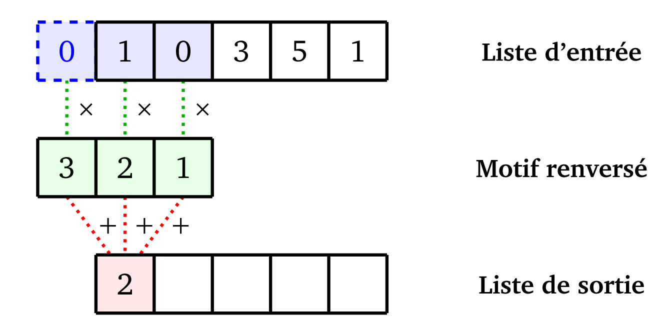
$$s = xc + yb + za$$

Pour le calcul des coefficients sur les bords, on rajoute des zéros virtuels à gauche et à droite de la liste d'entrée.

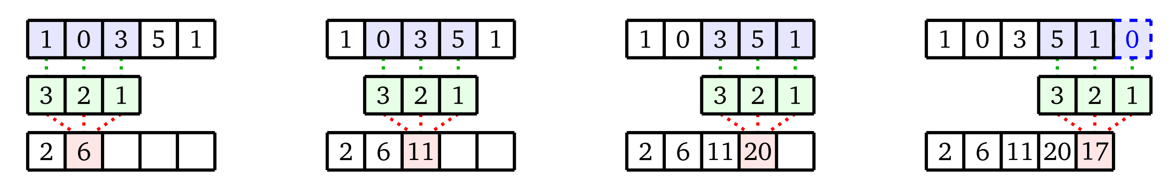


Notation. On note $f \star g$ ce ***produit de convolution tronqué***. La liste de sortie est de même taille que celle de la liste d'entrée.

Exemple.
Calculons la convolution $[1,0,3,5,1] \star [1,2,3]$. On commence par calculer le coefficient le plus à gauche (après ajout d'un zéro virtuel à la liste d'entrée).

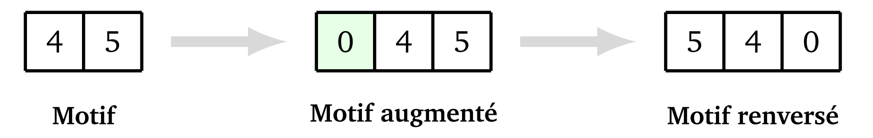


On continue en calculant les coefficients un par un.

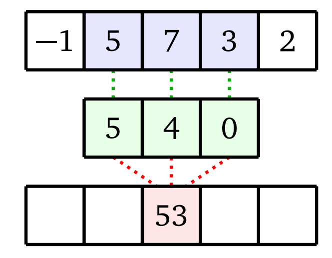


Ainsi $[1,0,3,5,1] \star [1,2,3] = [2,6,11,20,17]$.

Exemple.
Si le motif est de longueur paire alors on ajoute un 0 au début du motif pour le calcul. Ainsi $[-1,5,7,3,2] \star [4,5] = [-1,5,7,3,2] \star [0,4,5]$



Lorsque l'on renverse le motif, le zéro ajouté se retrouve à droite !



Après calcul on obtient $[-1,5,7,3,2] \star [4,5] = [-4,15,53,47,23]$.

2. Exemples

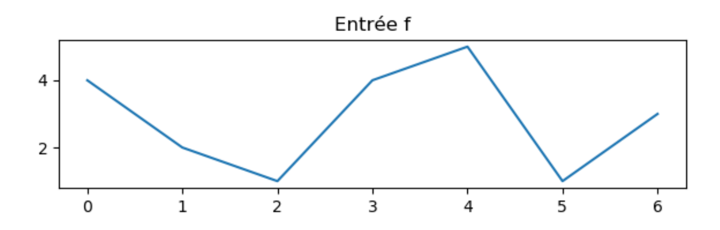
Voyons des exemples de motifs et leur effet sur des tableaux.

Moyenne mobile. La convolution par le motif $[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$ correspond à effectuer une moyenne mobile sur trois termes.

Exemple :

$$[4,2,1,4,5,1,3] \star \frac{1}{3}[1,1,1] \simeq [2.00,2.33,2.33,3.33,3.33,3.00,1.33]$$

On peut représenter graphiquement une liste $[x_0, x_1, \ldots, x_N]$ en reliant les points de coordonnées (k, x_k) pour k entre 0 et N. Voici la représentation du tableau d'entrée en bleu et celle du produit de convolution en rouge.



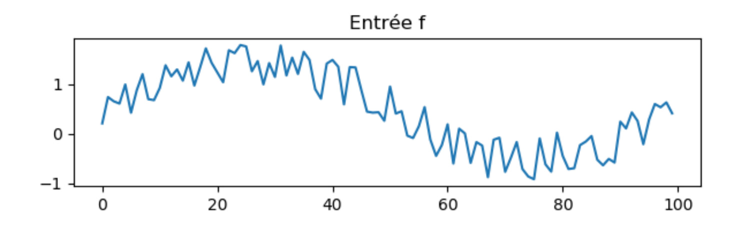


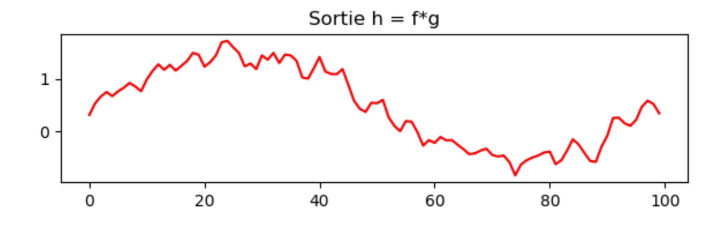
Plus généralement, le motif de longueur n : $\frac{1}{n}[1,1,1,\ldots,1]$ correspond à une moyenne mobile sur n termes. Cela permet de « lisser » des données et de supprimer du « bruit ».

Exemple :

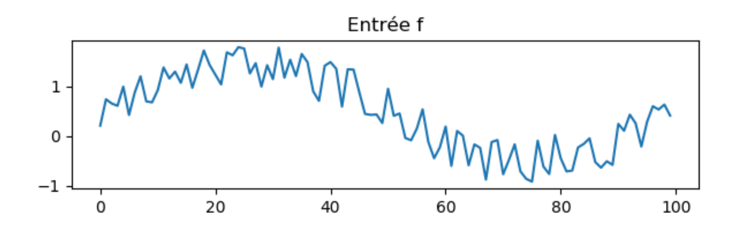
$$[4,2,1,4,5,1,3] \star \frac{1}{5}[1,1,1,1,1] = [1.4,2.2,3.2,2.6,2.8,2.6,1.8]$$

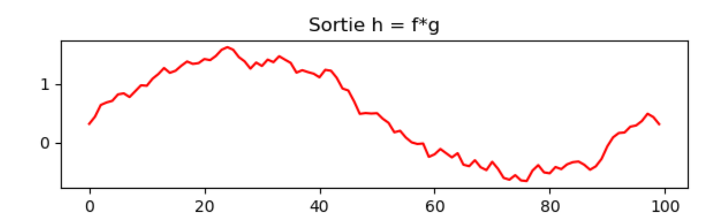
Voici la représentation d'un tableau de 100 points, au comportement assez chaotique, ainsi que le résultat de son produit de convolution par le motif $\frac{1}{3}[1,1,1]$: faire une moyenne mobile « lisse » la courbe.





En changeant maintenant le motif de convolution en $\frac{1}{5}[1,1,1,1,1]$ la courbe est encore plus lisse.





Homothétie. La convolution par le motif $[k]$ (de longueur 1 et qui pourrait encore s'écrire $[0,k,0]$) multiplie tous les termes de la liste d'entrée par un facteur k.
Exemple :

$$[1,2,3,4,5,6,7,8,9] \star [2] = [2,4,6,8,10,12,14,16,18].$$

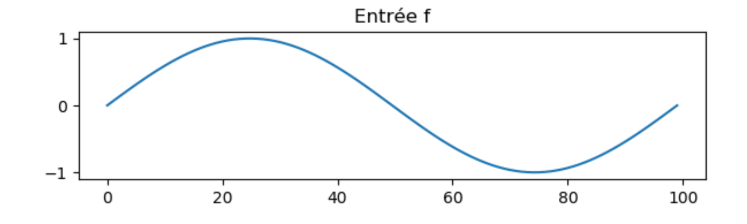
Translation. La convolution par le motif $[0,0,1]$ décale la liste d'un cran vers la droite (avec ajout d'un zéro en début, le dernier terme disparaissant).
Exemple :

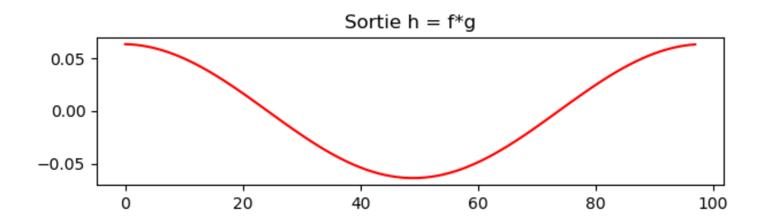
$$[1,2,3,4,5,6,7,8,9] \star [0,0,1] = [0,1,2,3,4,5,6,7,8].$$

Dérivée. La convolution par le motif $[0,1,-1]$ (qui s'écrit aussi $[1,-1]$) correspond au calcul de la dérivée de la liste d'entrée vu comme une fonction $n \mapsto f(n)$.
Exemple :

$$[16,9,4,1,0,1,4,9,16] \star [0,1,-1] = [16,-7,-5,-3,-1,1,3,5,7].$$

Le calcul correspond à $f(n+1)-f(n)$, une analogie discrète du taux d'accroissement $\frac{f(x+h)-f(x)}{h}$ avec $h=1$. Pour un tableau d'entrée de 100 valeurs $[\sin(0), \sin(2\pi/100), \sin(4\pi/100), \ldots, \sin(198\pi/100)]$, définies par la fonction sinus sur $[0,2\pi]$, le produit de convolution par le motif $[0,1,-1]$ donne logiquement un graphe ressemblant au cosinus (à un facteur multiplicatif près).

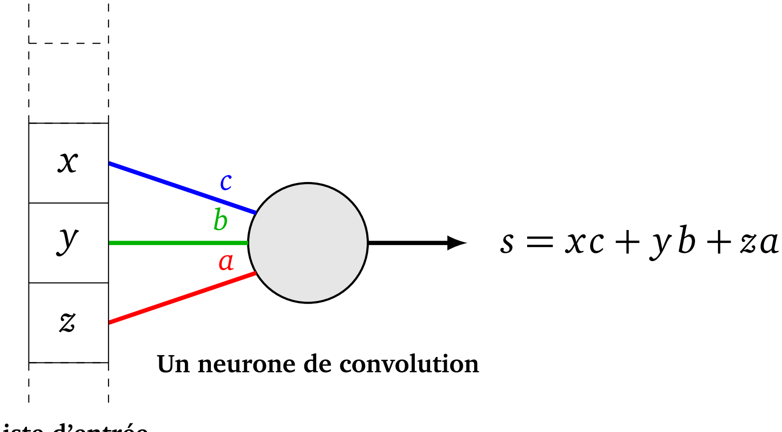




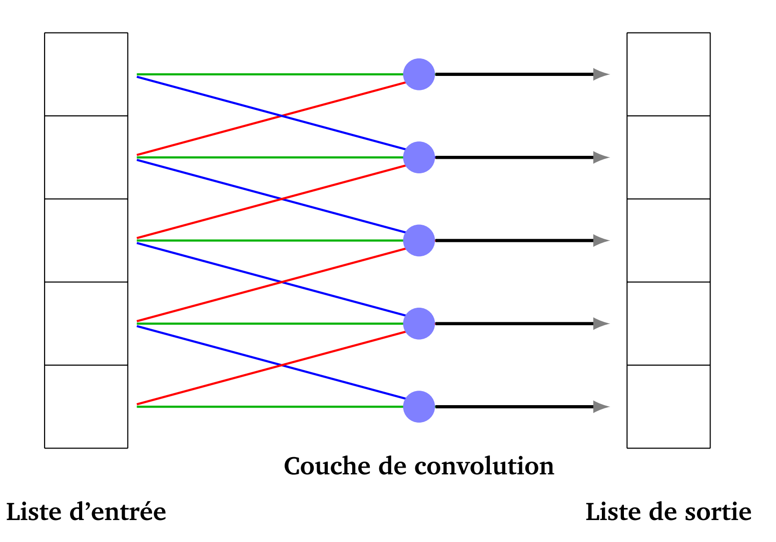
On retient donc que la convolution par un motif, peut rendre la liste plus lisse, la transformer (homothétie, translation) et peut en extraire certaines propriétés (comme la dérivée).

3. Réseau de neurones

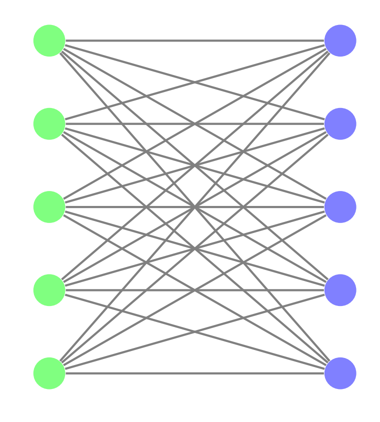
Comment créer un nouveau type de réseau de neurones réalisant un produit de convolution ? Nous allons créer un nouveau type de neurone et un nouveau type de couche de neurones correspondant à un motif donné.
Pour un motif $[a, b, c]$, un neurone de convolution possède trois arêtes, de poids a, b, c, et est relié à seulement trois valeurs de l'entrée.



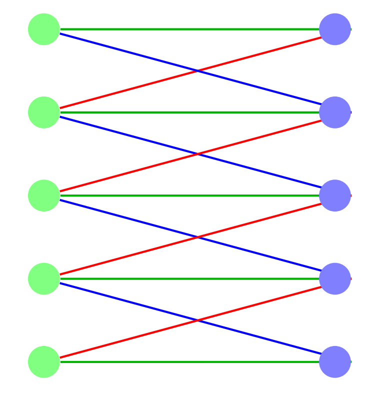
La sortie est $s = xc + yb + za$. On pourrait aussi composer par une fonction d'activation ϕ, auquel cas la sortie serait $s = \phi(xc + yb + za)$.
On peut représenter une couche de convolution par un ensemble de neurones de convolution, ceux-ci ayant tous les mêmes poids $[a, b, c]$ et chacun étant relié à seulement trois valeurs de l'entrée. La sortie produite est alors le produit convolution de l'entrée par le motif $[a, b, c]$. Pour une entrée de longueur n, la couche de convolution possède n neurones, chaque neurone ayant trois arêtes (sauf les neurones de bord). Mais au total, il n'y a que trois poids différents.



Quel est l'avantage de cette couche de neurones par rapport à une couche complètement connectée (comme dans les chapitres précédents) ? Dans une couche complètement connectée de n neurones ayant une entrée de taille n, il y a n^2 coefficients alors qu'ici il n'y a que 3 coefficients (quel que soit n). Par exemple, si $n = 100$ alors il y a seulement 3 coefficients à calculer au lieu de 10 000.



Couche complètement connectée



Couche de convolution

Sur la figure du gauche, nous avons une couche complètement connectée à la précédente. Chaque arête correspond à un poids à calculer. Sur la figure de droite, une couche de convolution (de taille 3) est connectée à la précédente. Chaque neurone n'a que 3 arêtes (sauf les neurones extrêmes qui n'en ont que deux) et de plus les arêtes partagent leur poids (les arêtes rouges ont toutes le même poids a, les arêtes vertes le même poids b, les arêtes bleues le même poids c).

4. Définition mathématique

Cette partie n'est pas utile pour les chapitres suivants, mais présente les choses de façon plus théorique. En particulier, il n'y a plus à ajouter de zéros virtuels (ni sur la liste d'entrée, ni sur un motif de longueur paire).

Définition.
Soient $(f(n))_{n\in\mathbb{Z}}$ et $(g(n))_{n\in\mathbb{Z}}$ deux suites de nombres réels. Le ***produit de convolution*** $f \mathbin{\tilde{\star}} g$ est la suite $(h(n))_{n\in\mathbb{Z}}$ dont le terme général est défini par :

$$\boxed{f \mathbin{\tilde{\star}} g(n) = \sum_{k=-\infty}^{+\infty} f(n-k)\cdot g(k)}$$

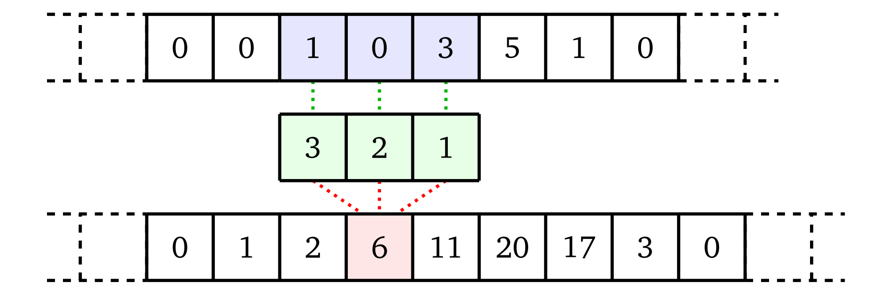
Il ne faut pas avoir peur de cette formule. En particulier, dans les situations rencontrées ici, il n'y a pas vraiment une infinité de termes à calculer. Voici une formule plus simple, lorsque l'on suppose que les termes de g sont nuls en dehors des indices appartenant à $[-K,+K]$:

$$\boxed{f \mathbin{\tilde{\star}} g(n) = \sum_{k=-K}^{+K} f(n-k)\cdot g(k)}$$

Reprenons l'exemple de la convolution de $[1,0,3,5,1]$ par $[1,2,3]$. Choisissons $(f(n))_{n\in\mathbb{Z}}$ la suite dont les termes non tous nuls sont $[f(0)=1, f(1)=0, f(2)=3, f(3)=5, f(4)=1]$ et choisissons $(g(n))_{n\in\mathbb{Z}}$ la suite dont les termes non tous nuls sont $[g(-1)=1, g(0)=2, g(1)=3]$. Alors on obtient la convolution $h = f \mathbin{\tilde{\star}} g$ par la formule ci-dessus :

$$f \mathbin{\tilde{\star}} g(n) = \sum_{k=-1}^{+1} f(n-k)\cdot g(k) = f(n-1)g(1) + f(n)g(0) + f(n+1)g(-1).$$

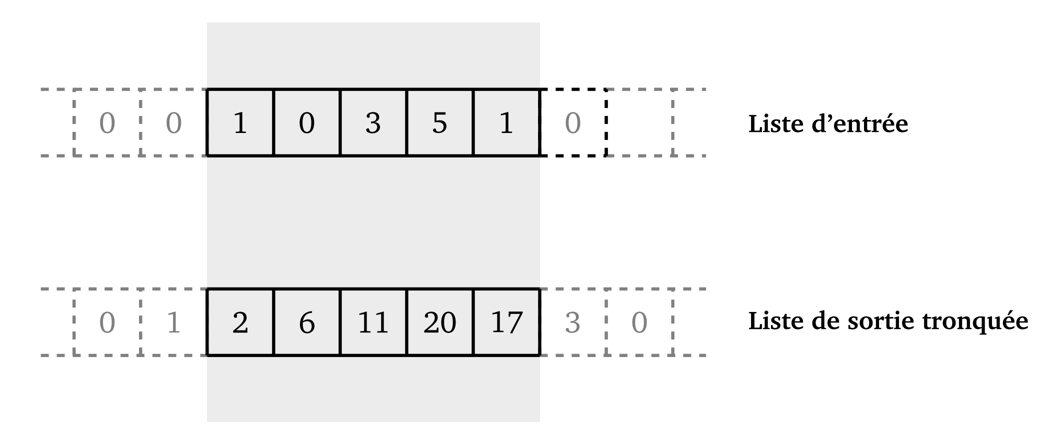
On retrouve la méthode pratique vu précédemment : on renverse le motif g avant de faire une série de multiplications. Par exemple $f \mathbin{\tilde{\star}} g(1) = f(0)g(1) + f(1)g(0) + f(2)g(-1) = 1\times 3 + 0\times 2 + 3\times 1 = 6$.



Le résultat de la convolution est la suite

$$[\ldots, 0, 0, 1, 2, 6, 11, 20, 17, 3, 0, 0, \ldots]$$

où le terme de rang 0 est $f \mathbin{\tilde{\star}} g(0) = 2$. Ce résultat est indépendant, à un décalage près, des choix opérés dans les définitions de f et g.
La convolution tronquée consiste à ne retenir qu'une sous-liste de même taille que la liste d'entrée en tronquant les bords (la façon de tronquer dépend du choix des définitions de f et g). Ainsi : $[1, 0, 3, 5, 1] \star [1, 2, 3] = [2, 6, 11, 20, 17]$.



Terminons par une propriété remarquable du produit de convolution. En réindexant la somme de la définition, on obtient :

$$f \mathbin{\tilde{\star}} g(n) = \sum_{k=-\infty}^{+\infty} f(k) \cdot g(n-k).$$

Ce qui signifie que le produit de convolution est symétrique :

$$\boxed{f \mathbin{\tilde{\star}} g(n) = g \mathbin{\tilde{\star}} f(n)}$$

Remarque.

- La convolution est très utilisée en théorie du signal, c'est pourquoi on trouve aussi le vocabulaire *signal d'entrée*, *signal de sortie*. C'est aussi pour des raisons de signal qu'on renverse une des listes avant de faire les multiplications.
- Le ***motif*** s'appelle aussi ***noyau*** (*kernel* en anglais) mais aussi ***masque*** ou ***filtre***.

Chapitre 12

Convolution

Vidéo ■ partie 12.1. Convolution d'une matrice
Vidéo ■ partie 12.2. Neurone et couche de convolution
Vidéo ■ partie 12.3. Les différentes couches d'un réseau
Vidéo ■ partie 12.4. Traitement des images

La convolution est une opération mathématique simple sur un tableau de nombres, une matrice ou encore une image afin d'y apporter une transformation ou d'en tirer des caractéristiques principales.

1. Matrice

Nous allons voir ou revoir quelques notions (le minimum vital) sur les matrices.

1.1. Vocabulaire

- Une ***matrice*** est un tableau de n lignes et p colonnes, contenant des nombres réels.

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{np} \end{pmatrix}}_{p \text{ colonnes}} \quad n \text{ lignes}$$

- Lorsque il y a autant de lignes que de colonnes, on parle de ***matrice carrée***. Voici une matrice 3×3 :

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

- ***Addition.*** On peut additionner deux matrices de même taille. L'addition se fait terme à terme. Exemple :

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 3 \\ 5 & 1 & 7 \\ 7 & 9 & 9 \end{pmatrix}$$

- ***Multiplication par un scalaire.*** On peut multiplier une matrice par un nombre réel. Exemple :

$$3 \cdot \begin{pmatrix} 1 & 2 & 3 & 4 \\ 8 & 7 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 3 & 6 & 9 & 12 \\ 24 & 21 & 15 & 18 \end{pmatrix}$$

- ***Multiplication.*** On peut multiplier deux matrices. Nous ne donnons pas ici la définition, mais la multiplication *n'est pas* effectuée terme à terme. Voici, sans entrer dans les détails, un exemple pour des

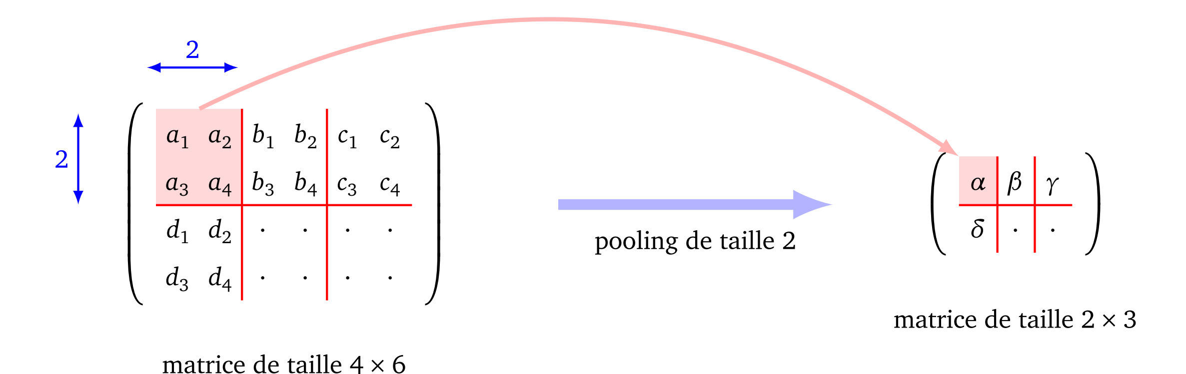
matrices carrées :

$$\begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} -6 & -9 & -12 \\ 10 & 11 & 12 \\ 18 & 21 & 24 \end{pmatrix}$$

Nous définirons un peu plus loin la convolution qui est un nouveau type de multiplication de matrices.

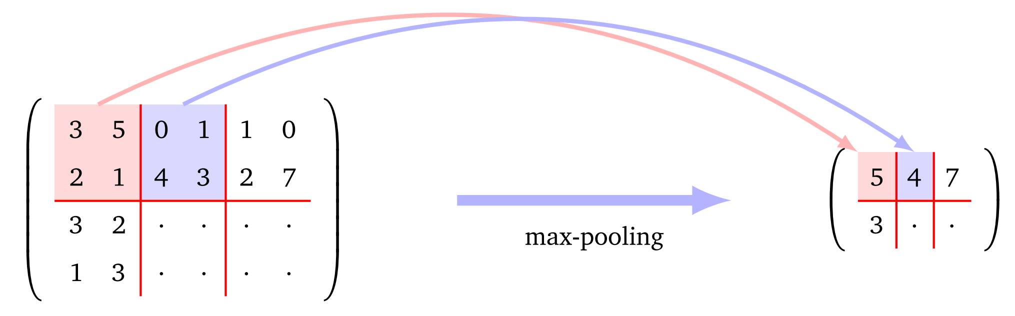
1.2. Pooling

Le *pooling* (regroupement de termes) consiste à transformer une matrice en une matrice plus petite tout en essayant d'en garder les caractéristiques principales.
Un ***pooling*** de taille k transforme une matrice de taille $n \times p$ en une matrice de taille k fois plus petite, c'est-à-dire de taille $n//k \times p//k$. Une sous-matrice de taille $k \times k$ de la matrice de départ produit un seul coefficient de la matrice d'arrivée.

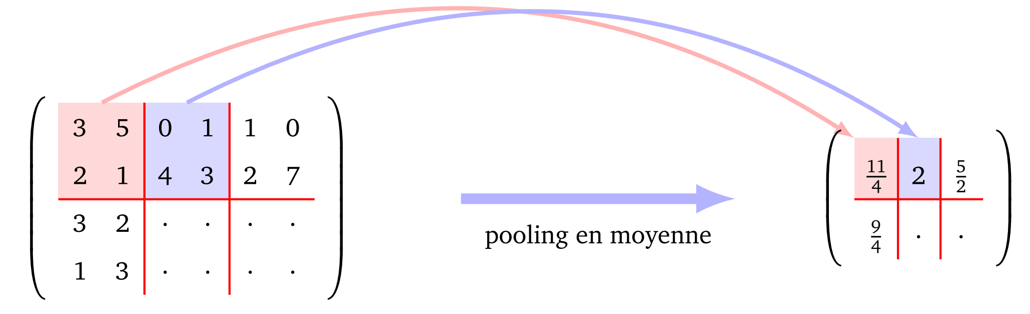


On distingue deux types de pooling.

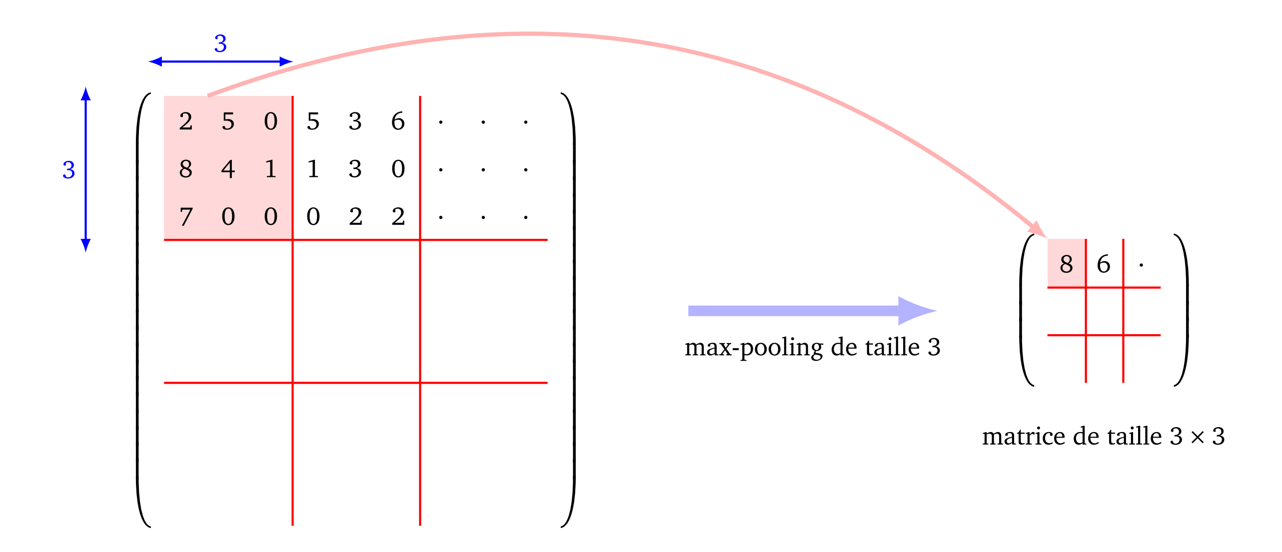
- Le ***max-pooling*** de taille k consiste à retenir le maximum de chaque sous-matrice de taille $k \times k$:



- Le ***pooling en moyenne*** de taille k (*average pooling*) consiste à retenir la moyenne des termes de chaque sous-matrice de taille $k \times k$:



Ci-dessous, voici le début d'un max-pooling de taille 3. La matrice de départ de taille 9×9 est transformée en une matrice de taille 3×3.



matrice de taille 9 × 9

Le max-pooling, qui ne retient que la valeur la plus élevée par sous-matrice, permet de détecter la présence d'une caractéristique (par exemple un pixel blanc dans une image noire). Tandis que le pooling en moyenne prend en compte tous les termes de chaque sous-matrice (par exemple avec 4 pixels d'une image de ciel, on retient la couleur moyenne).

2. Convolution : deux dimensions

2.1. Définition

La convolution en deux dimensions est une opération qui :

- à partir d'une matrice d'entrée notée A,
- et d'une matrice d'un motif noté M,

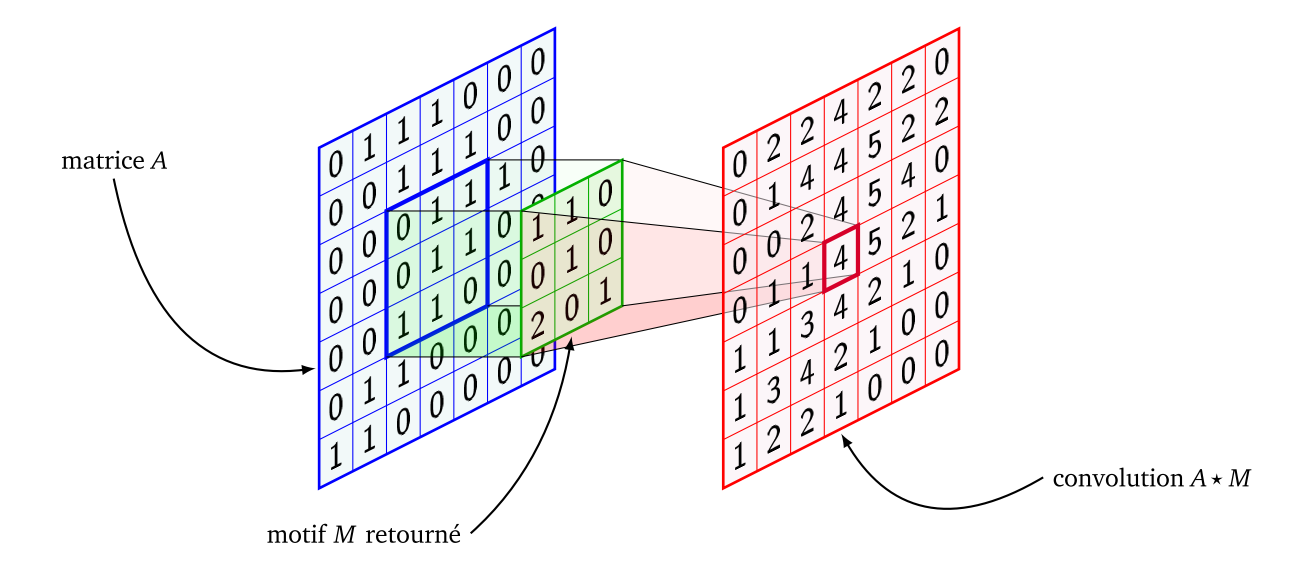
associe une matrice de sortie $A \star M$.
Tout d'abord, il faut retourner la matrice M :

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \longrightarrow \begin{pmatrix} m_{33} & m_{32} & m_{31} \\ m_{23} & m_{22} & m_{21} \\ m_{13} & m_{12} & m_{11} \end{pmatrix}$$

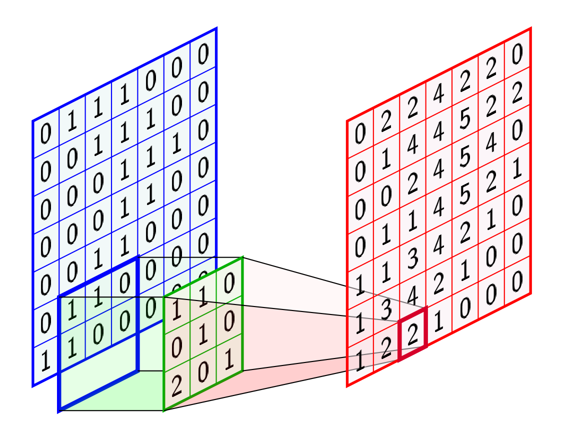
Motif de taille 3×2 **Motif retourné**

Le calcul de $A \star M$ s'effectue coefficient par coefficient :

- on centre le motif retourné sur la position du coefficient à calculer,
- on multiplie chaque coefficient de A par le coefficient du motif retourné en face (quitte à ajouter des zéros virtuels sur les bords de A),
- la somme de ces produits donne un coefficient de $A \star M$.



Voici un autre schéma pour le calcul d'un autre coefficient. Pour ce calcul, on rajoute des zéros virtuels autour de la matrice A.



Exemple.
Calculons la convolution $A \star M$ définie par :

$$\begin{pmatrix} 2 & 1 & 3 & 0 \\ 1 & 1 & 0 & 5 \\ 3 & 3 & 1 & 0 \\ 2 & 0 & 0 & 2 \end{pmatrix} \star \begin{pmatrix} 1 & 0 & 2 \\ 2 & 1 & 0 \\ 1 & 0 & 3 \end{pmatrix}.$$

On commence par retourner M. Pour calculer le premier coefficient de $A \star M$, on centre le motif sur le premier coefficient de A, puis on rajoute des zéros virtuels à gauche et en haut. Ensuite on calcule les produits des coefficients de la matrice M retournée avec les coefficients de A correspondants, et on les additionne :

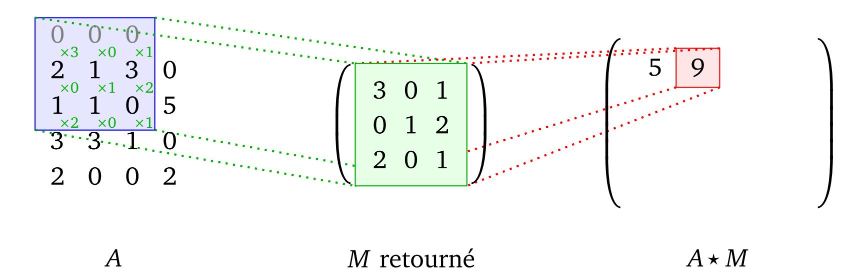
$$0 \times 3 + 0 \times 0 + 0 \times 1 + 0 \times 0 + 2 \times 1 + 1 \times 2 + 0 \times 2 + 1 \times 0 + 1 \times 1.$$

Cette somme vaut 5, c'est le premier coefficient de $A \star M$.

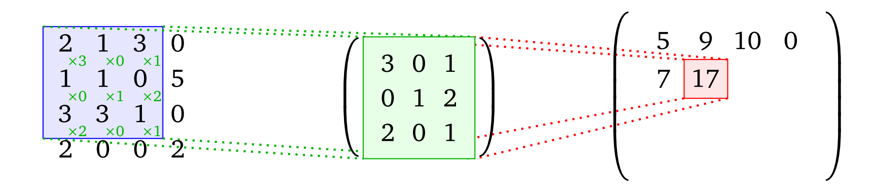
$$\begin{matrix} 0 & 0 & 0 & & \\ 0 & 2 & 1 & 3 & 0 \\ 0 & 1 & 1 & 0 & 5 \\ & 3 & 3 & 1 & 0 \\ & 2 & 0 & 0 & 2 \end{matrix} \qquad \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 2 \\ 2 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 5 & & & \\ & & & \\ & & & \\ & & & \end{pmatrix}$$

A $\qquad$ M retourné $\qquad$ $A \star M$

On continue avec le second coefficient.



Et ainsi de suite :



Jusqu'à calculer entièrement la matrice $A \star M$.

$$A = \begin{matrix} 2 & 1 & 3 & 0 \\ 1 & 1 & 0 & 5 \\ 3 & 3 & 1 & 0 \\ 2 & 0 & 0 & 2 \end{matrix} \qquad \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 2 \\ 2 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 5 & 9 & 10 & 0 \\ 7 & 17 & 19 & 16 \\ 10 & 12 & 11 & 0 \\ 5 & 10 & 13 & 5 \end{pmatrix}$$

Remarque.

- Il ne faut pas confondre le fait de retourner la matrice avec une opération différente qui s'appelle la transposition.
- Dans la plupart de nos situations, le motif sera une matrice de taille 3×3. Par contre la matrice A pourra être de très grande taille (par exemple une matrice 600×400, codant une image).
- Cependant, si la matrice du motif possède une dimension paire on rajoute des zéros virtuels à gauche ou en haut (avant de la retourner).

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \longrightarrow \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 3 & 4 \end{pmatrix} \longrightarrow \begin{pmatrix} 4 & 3 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Motif de taille 2×2 **Motif augmenté** **Motif retourné**

- Contrairement au pooling, les motifs disposés pour des calculs d'éléments voisins se superposent. Sur le dessin ci-dessous le motif est d'abord centré sur le coefficient 8, puis sur le coefficient 9.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ & & & & & \\ & & & & & \end{pmatrix}$$

- On verra dans les chapitres « Convolution avec Python » et « Convolution avec tensorflow/keras » plus de paramètres pour la convolution (comment déplacer le motif, quel choix faire au bord de la matrice A, etc.).
- Par définition, il faut retourner la matrice M avant de calculer les produits. Cela complique l'usage mais est cohérent avec la définition donnée pour la dimension 1 et cela permettra d'avoir de bonnes propriétés mathématiques (voir le chapitre « Convolution avec Python »).

2.2. Exemples simples

Exemple 3×3**.** Voici un autre exemple de calcul :

$$\begin{pmatrix} 2 & -1 & 7 & 3 & 0 \\ 2 & 0 & 0 & -2 & 1 \\ -5 & 0 & -1 & -1 & 4 \end{pmatrix} \star \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

Il ne faut pas oublier de retourner M avant de commencer !

$$\begin{matrix} 2 & -1 & 7 & 3 & 0 \\ 2 & 0 & 0_{\times 2} & -2_{\times 1} & 1_{\times 0} \\ -5 & 0 & -1_{\times 1} & -1_{\times 1} & 4_{\times -1} \\ & & 0_{\times 0} & 0_{\times -1} & 0_{\times -2} \end{matrix} \qquad \begin{pmatrix} 2 & 1 & 0 \\ 1 & 1 & -1 \\ 0 & -1 & -2 \end{pmatrix} \qquad \begin{pmatrix} 1 & -6 & 7 & 10 & 2 \\ 9 & 7 & 10 & 7 & 1 \\ -3 & 0 & 0 & -8 & 0 \end{pmatrix}$$

A — M retourné — $A \star M$

Translation. Une convolution par la matrice

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

correspond à une translation des coefficients vers le bas. Par exemple :

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \star \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}.$$

De même pour une translation des coefficients vers la droite.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \star \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 5 & 6 & 7 \\ 0 & 9 & 10 & 11 \\ 0 & 13 & 14 & 15 \end{pmatrix}.$$

Moyenne. On effectue une moyenne locale des coefficients à l'aide du motif :

$$\frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

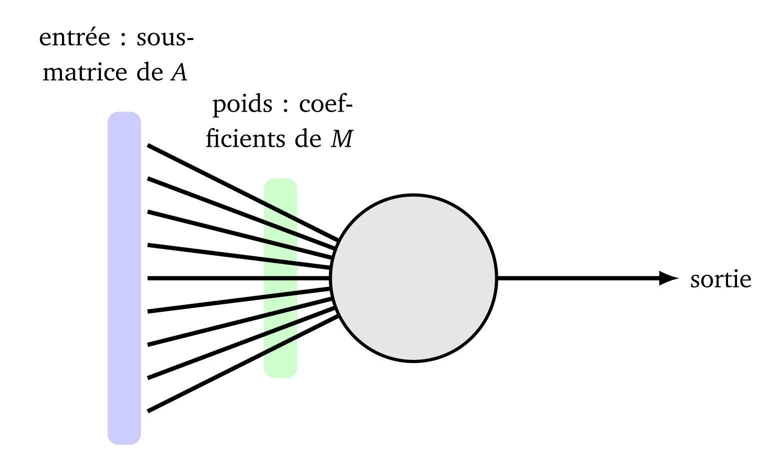
Par exemple :

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{pmatrix} \star \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \simeq \begin{pmatrix} 1.77 & 3 & 3.66 & 4.33 & 3.11 \\ 4.33 & 7 & 8 & 9 & 6.33 \\ 7.66 & 12 & 13 & 14 & 9.66 \\ 11 & 17 & 18 & 19 & 13 \\ 8.44 & 13 & 13.66 & 14.33 & 9.77 \end{pmatrix}.$$

On remarque des phénomènes de bords dus à l'ajout de zéros virtuels.

2.3. Neurone de convolution

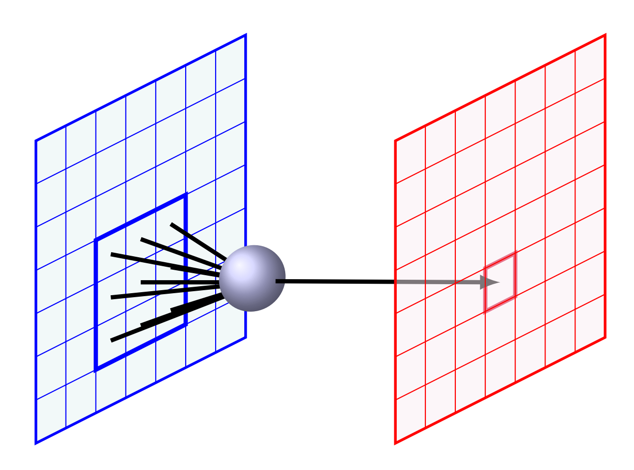
À l'aide de la convolution, nous allons définir un nouveau type de couche de neurones : une « couche de convolution ». Tout d'abord un ***neurone de convolution*** de taille 3×3 est un neurone classique ayant 9 entrées (pour l'instant la fonction d'activation est l'identité).



Les poids du neurone correspondent aux coefficients d'une matrice de convolution :

$$\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}.$$

Imaginons que l'entrée soit une image ou un tableau de dimension 2, pour nous ce sera une matrice A. Alors un neurone de convolution est relié à une sous-matrice 3×3 de A.



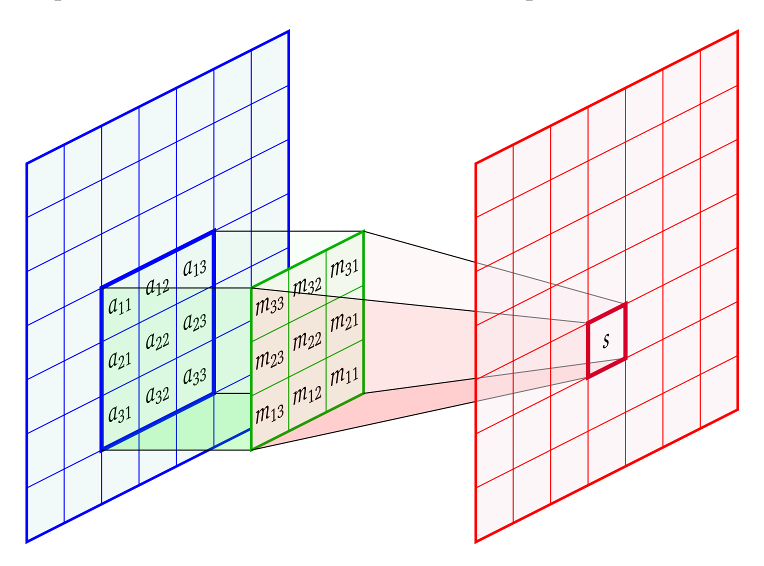
Ce neurone agit comme un élément de convolution. Par exemple si la sous-matrice de A est notée

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

alors le neurone produit la sortie :

$$s = a_{11} \cdot m_{33} + a_{12} \cdot m_{32} + a_{13} \cdot m_{31} + a_{21} \cdot m_{23} + a_{22} \cdot m_{22} + a_{23} \cdot m_{21} + a_{31} \cdot m_{13} + a_{32} \cdot m_{12} + a_{33} \cdot m_{11}$$

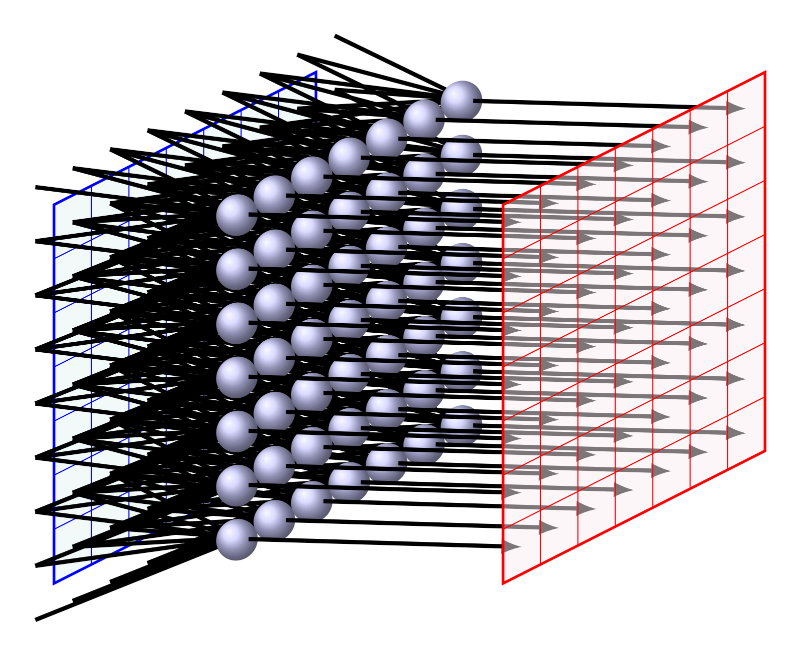
N'oublier pas que dans le produit de convolution, on commence par retourner la matrice M.



On pourrait aussi composer par une fonction d'activation au niveau du neurone, mais on le fera plus tard à l'aide d'une « couche d'activation ». Plus généralement un neurone de convolution de taille $p \times q$ possède pq arêtes et donc pq poids à définir ou à calculer.

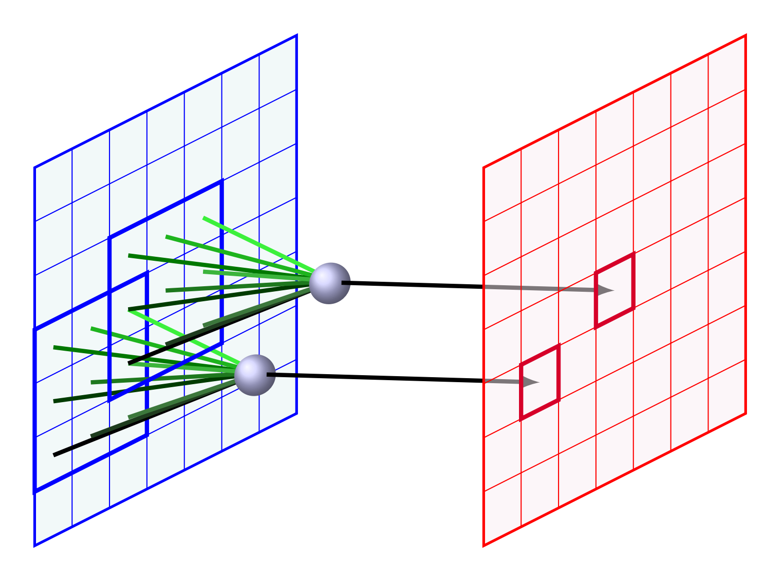
2.4. Couche de convolution

Considérons une entrée A représentée par une matrice de taille $n \times p$.
Une *couche de convolution* (pour un seul motif) est la donnée d'une matrice M appelé *motif* (par exemple de taille 3×3) et qui renvoie en sortie les coefficients de la matrice $A \star M$.



Pour une entrée de taille $n \times p$, il y a donc np neurones de convolutions, chacun ayant 9 arêtes (car le motif est de taille 3×3). Il est très important de comprendre que les poids sont communs à tous les neurones (ce sont les coefficients de M). Ainsi, pour une couche de convolution, il y a seulement 9 poids à déterminer pour définir la couche de convolution (bien que le nombre de neurones puisse être très grand).
Sur le dessin ci-dessous on ne montre que deux neurones. Noter deux choses importantes : (a) deux neurones différents peuvent avoir certaines entrées communes (sur le dessin les deux carrés 3×3 s'intersectent) et

(b) deux neurones différents ont les mêmes poids (sur le dessin les arêtes entrantes ayant la même couleur, ont les mêmes poids).



Remarque.

- Terminologie : le motif M s'appelle aussi le ***noyau*** (*kernel*), le ***filtre*** ou encore le ***masque***.
- Le motif peut être d'une taille différente de la taille 3×3 considérée dans les exemples qui est cependant la plus courante.
- Il existe également des variantes avec biais ou fonction d'activation.
- Combinatoire : dans le cas d'une couche complètement connectée, le nombre de poids à calculer serait énorme. En effet, une couche de np neurones complètement connectée à une entrée de taille $n \times p$ amènerait à calculer $(np)^2$ poids. Pour $n = p = 100$, cela fait 10 000 neurones et 100 000 000 poids. Pour rappel, notre couche de convolution est définie par 9 poids (quels que soient n et p).
- D'un point de vue mathématique, une couche de convolution associée au motif M est l'application :

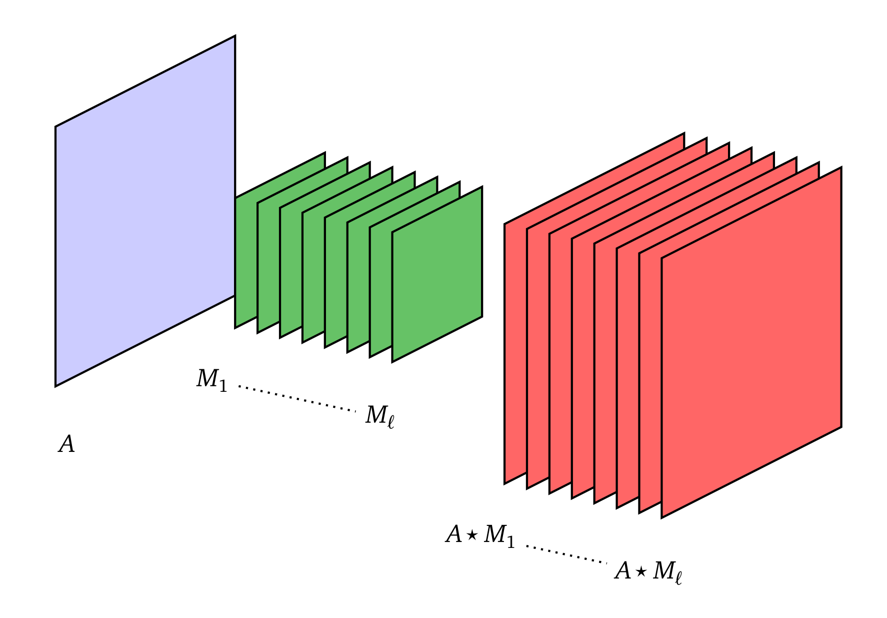
$$F : \begin{array}{ccc} \mathcal{M}_{n,p} & \longrightarrow & \mathcal{M}_{n,p} \\ A & \longmapsto & A \star M \end{array}$$

  où $\mathcal{M}_{n,p}$ est l'ensemble des matrices de taille $n \times p$.
  Si on transforme une matrice en un (grand) vecteur, alors on obtient une application $F : \mathbb{R}^{np} \to \mathbb{R}^{np}$.

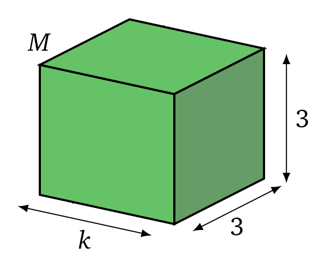
2.5. Différentes couches d'un réseau de neurones

Les couches de convolution sont au cœur des réseaux de neurones modernes. Voici un petit survol des types de couches qui seront mises en pratique dans le chapitre « Convolution avec tensorflow/keras ».

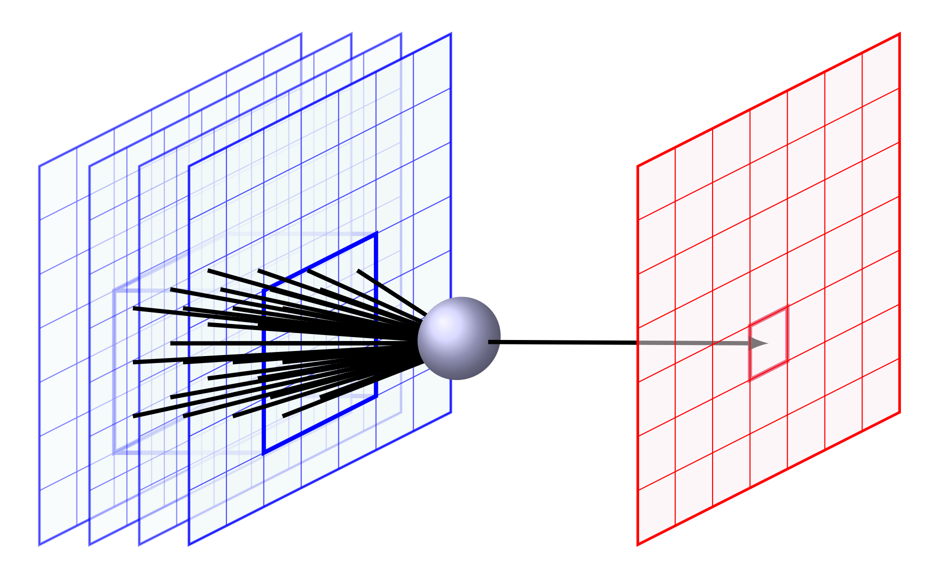
Plusieurs filtres. On verra dans la pratique qu'une couche de convolution est définie avec plusieurs motifs, c'est-à-dire un nombre ℓ de matrices $M_1, M_2, \ldots, M_\ell$. Ainsi pour une entrée A de taille $n \times p$, une couche de convolution à ℓ motifs renvoie une sortie de taille $n \times p \times \ell$, correspondant à $(A \star M_1, A \star M_2, \ldots, A \star M_\ell)$.



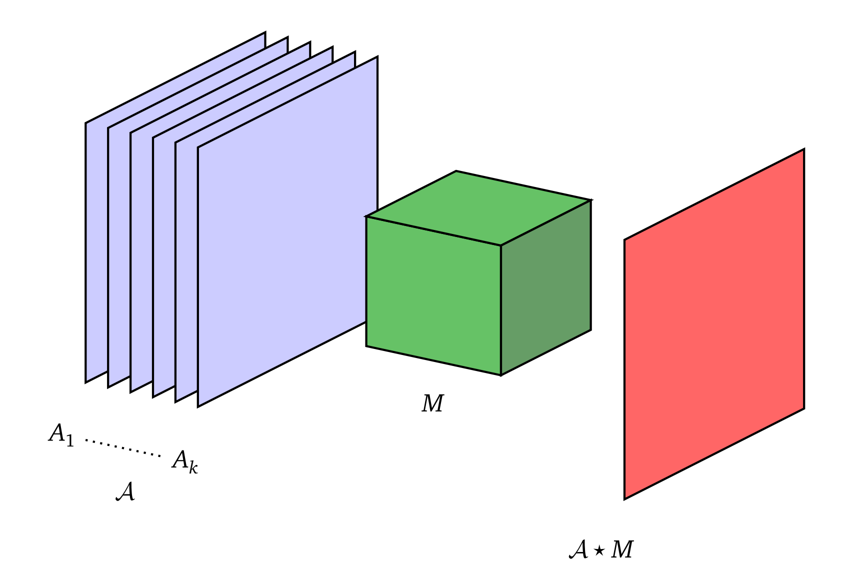
Convolution à partir de plusieurs canaux. Pour traiter une sortie de taille $n \times p \times k$ du type précédent, qui va être l'entrée de la couche suivante on doit définir une convolution ayant plusieurs canaux en entrée. Si les entrées sont les $A_1, A_2, \ldots, A_k$ alors un motif M associé à cette entrée de profondeur k est un 3-tenseur de taille $(3, 3, k)$, c'est-à-dire un tableau à 3 dimensions de la forme $3 \times 3 \times k$ (on renvoie au chapitre « Tenseurs » pour la définition).



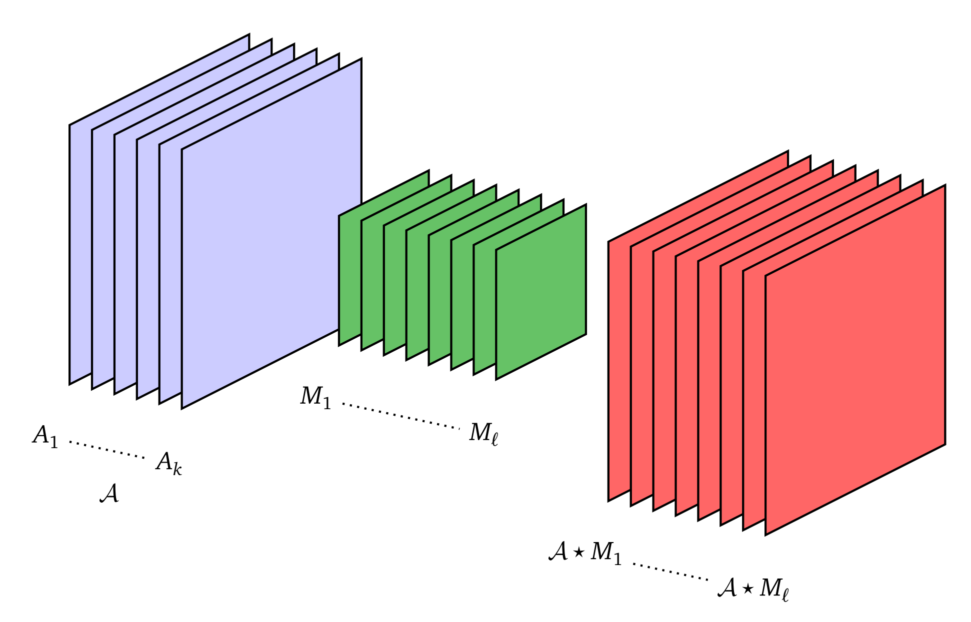
Ce motif M correspond donc à une couche de neurones où chaque neurone possède $3 \times 3 \times k$ arêtes. Chaque neurone est connecté localement à une zone 3×3 dans un plan, mais ceci sur toute la profondeur des k entrées. Le calcul de la sortie du neurone se fait en réalisant la somme de $3 \times 3 \times k$ termes. Pour réaliser une couche de neurones, la zone 3×3 se déplace dans le plan, mais s'étend toujours sur toute la profondeur.



On note $\mathcal{A} = (A_1, A_2, \ldots, A_k)$ et $\mathcal{A} \star M$ le produit de convolution. Si l'entrée $\mathcal{A}$ est de taille (n, p, k) alors la sortie $\mathcal{A} \star M$ associée au motif M est de taille (n, p).



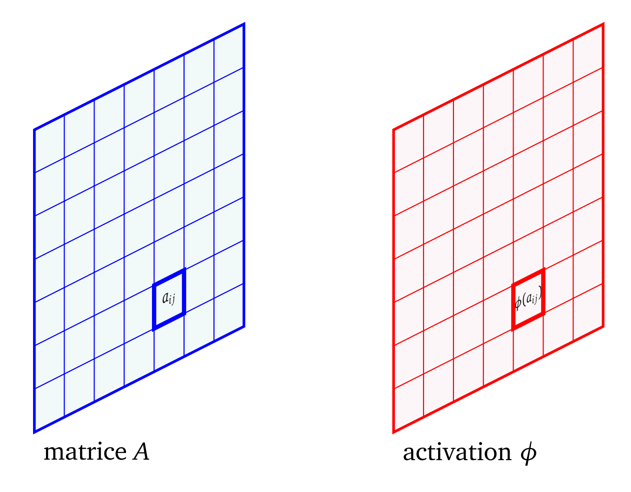
Convolution à plusieurs filtres à partir de plusieurs canaux. C'est le cas général dans la pratique. Une entrée donnée par plusieurs canaux $\mathcal{A} = (A_1, A_2, \ldots, A_k)$, associée à des motifs $M_1, M_2, \ldots, M_\ell$ (qui sont donc chacun des 3-tenseurs de taille $(3,3,k)$) produit une sortie de taille $n \times p \times \ell$, correspondant à $(\mathcal{A} \star M_1, \mathcal{A} \star M_2, \ldots, \mathcal{A} \star M_\ell)$. Si l'entrée $\mathcal{A}$ est de taille (n, p, k) alors la sortie est de taille (n, p, ℓ).



Noter que sur cette figure chaque motif M_i est représenté par carré 3×3, alors qu'en fait chacun devrait être une boîte en 3 dimensions de taille $3 \times 3 \times k$.

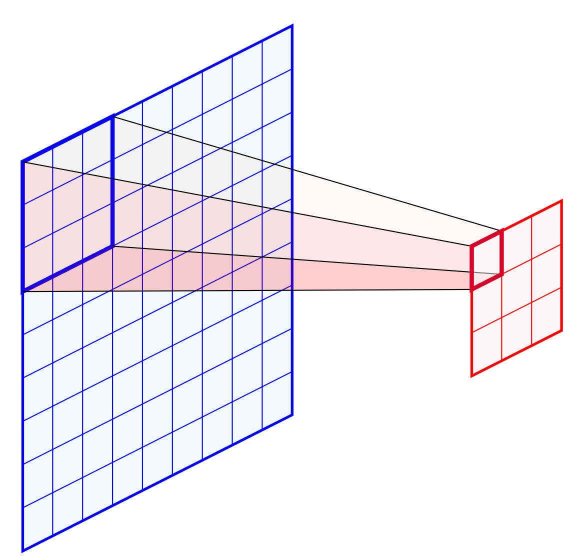
Activation. On peut composer par une fonction d'activation ϕ directement dans le neurone ou bien dans une couche à part (ce qui est équivalent). Comme d'habitude, la fonction d'activation est la même pour tous les neurones d'une couche. Ainsi une ***couche d'activation*** pour la fonction d'activation ϕ, transforme une entrée

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1p} \\ a_{21} & a_{22} & \ldots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{np} \end{pmatrix} \quad \text{en} \quad \begin{pmatrix} \phi(a_{11}) & \phi(a_{12}) & \ldots & \phi(a_{1p}) \\ \phi(a_{21}) & \phi(a_{22}) & \ldots & \phi(a_{2p}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(a_{n1}) & \phi(a_{n2}) & \ldots & \phi(a_{np}) \end{pmatrix}.$$

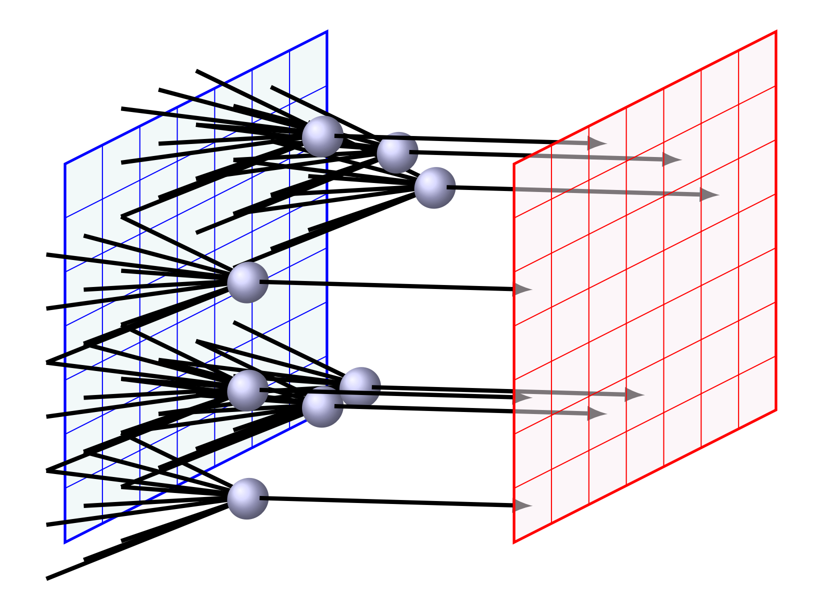


Les fonctions d'activation sont les mêmes que celles rencontrées auparavant : ReLU, σ, tanh...

Pooling. Une couche de pooling de taille k transforme une entrée de taille $n \times p$ en une sortie de taille $n//k \times p//k$. Nous renvoyons à la première section de ce chapitre pour le max-pooling ou le pooling en moyenne. Cette opération permet de réduire les données d'un facteur k^2. Par exemple, une entrée de taille 100×80 (8000 données) avec un pooling de taille 4 fournit une sortie de taille 25×20 (500 données). Ci-dessous un pooling de taille 3 transforme une matrice 9×9 en une matrice 3×3.



Dropout. Le *dropout* (décrochage ou abandon en français) est une technique pour améliorer l'apprentissage d'un réseau et en particulier pour prévenir le sur-apprentissage. L'idée est de désactiver certains neurones d'une couche lors des étapes de l'apprentissage. Ces neurones sont choisis au hasard et sont désactivés temporairement pour une itération (par exemple on peut choisir à chaque itération de désactiver un neurone avec une probabilité $\frac{1}{2}$). Cela signifie que l'on retire toute arête entrante et toute arête sortante de ces neurones, ce qui revient à mettre les poids à zéro tant pour l'évaluation que pour la rétropropagation. Lors de l'itération, suivante on choisit de nouveau au hasard les neurones à désactiver. Voir le chapitre « Probabilités » pour plus de détails.



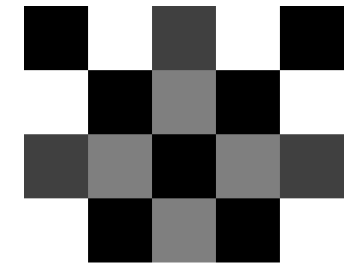
3. Traitement des images

La convolution est un outil puissant dans le domaine de la reconnaissance d'images et est indispensable dans les réseaux de neurones modernes. Pour mieux comprendre ce que permet la convolution, nous allons voir comment certains motifs transforment les images.

Dans cette section, une image en niveaux de gris est une matrice ayant ses entrées entre 0 et 255. Chaque entrée de la matrice représente un pixel, 0 pour un pixel noir, 255 pour un pixel blanc, les valeurs intermédiaires étant des nuances de gris.

Voici un petit exemple.

$$A = \begin{pmatrix} 0 & 255 & 64 & 255 & 0 \\ 255 & 0 & 127 & 0 & 255 \\ 64 & 127 & 0 & 127 & 64 \\ 255 & 0 & 127 & 0 & 255 \end{pmatrix}$$



C'est sur une image/matrice A que l'on va appliquer une convolution.

3.1. Flou

On obtient une image floue par application du motif :

$$M = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$



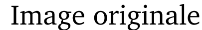


Image originale



Flou

En effet, la convolution par cette matrice M remplace la couleur d'un pixel par la moyenne de la couleur de ses 9 pixels voisins. Outre l'aspect esthétique, il y a un intérêt fondamental : le flou réduit le bruit. Par exemple, si un pixel est mauvais (par exemple à cause d'une erreur de l'appareil photo), le flou élimine cette erreur.
Une variante est le *flou gaussien* défini par la matrice :

$$M = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}.$$

Il existe aussi des flous définis par des matrices M de taille 5×5, qui tiennent compte des 25 pixels voisins.

3.2. Piqué

C'est en quelque sorte le contraire du flou ! Le motif est :

$$M = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}.$$

Voici un exemple avec l'image originale à gauche et l'image transformée à droite qui a l'air plus nette que l'originale !



Image originale



Piqué

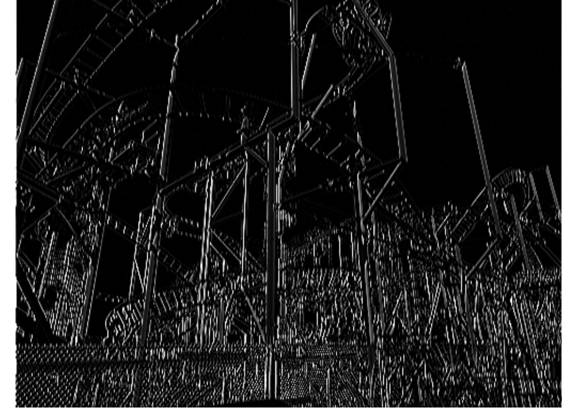
3.3. Verticales et horizontales

Le motif suivant renforce les lignes verticales :

$$M = \begin{pmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{pmatrix}.$$



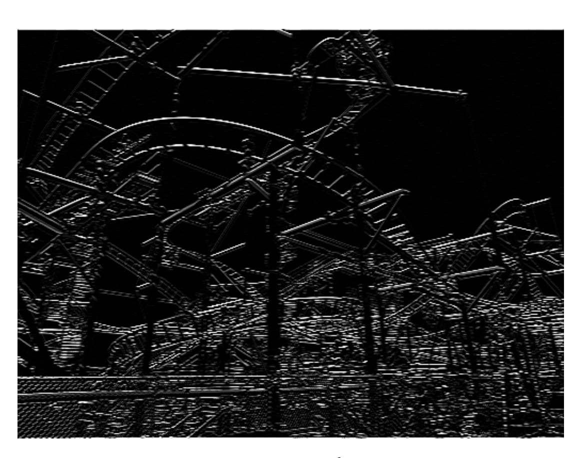
Image originale



Verticales

De même pour les lignes horizontales (à gauche) ou les lignes à 45° (à droite) :

$$M = \begin{pmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{pmatrix}, \qquad M = \begin{pmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{pmatrix}.$$



Horizontales



Diagonales

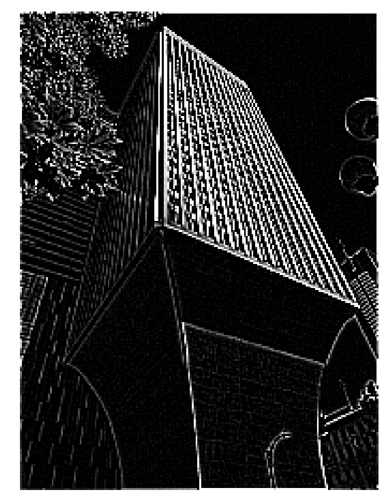
Ce que l'on retient, c'est que la convolution permet de détecter des lignes verticales ou horizontales par exemple et donc d'extraire des caractéristiques abstraites d'une image.

3.4. Bords

On peut plus généralement extraire les bords des formes d'une image avec :

$$M = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}.$$


Image originale


Contour

Des variantes sont les matrices :

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad \text{ou} \qquad M = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{pmatrix}.$$

3.5. Mise en relief

En jouant sur les contrastes, on peut créer une impression de relief avec :

$$M = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$


Image originale


Relief

3.6. Pooling

On termine cette section en illustrant l'action du pooling sur une image.



Image originale





Max-pooling 4×4



Pooling 4×4 en moyenne

Les images obtenues comportent 16 fois moins de pixels. Les voici agrandies.





Max-pooling 4×4



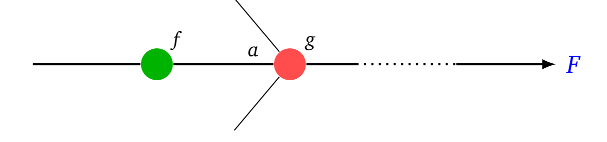
Pooling 4×4 en moyenne

4. Rétropropagation

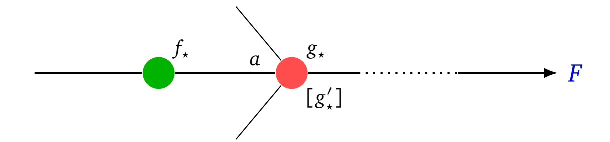
Les formules de la rétropropagation s'étendent sans problème au cas des neurones de convolution. On commence par reprendre les calculs du chapitre « Gradient » pour le cas classique avant de s'occuper du cas de la convolution.

Cas classique.

Voici la situation d'un neurone. On ne s'intéresse qu'à une seule de ses arêtes d'entrée : celle associée au poids a et de valeur d'entrée f. Ce neurone a pour fonction d'activation g. La sortie de ce neurone est utilisée par le reste du réseau. À la fin, on obtient une valeur de sortie pour la fonction F.



On distingue la fonction de sa valeur en un point : on note f la fonction et $f_\star$ la valeur de la fonction à la sortie du neurone correspondant. De même pour g et $g_\star$ et la valeur de la dérivée $g'_\star$.



On reprend la formule du chapitre « Gradient » :

$$\frac{\partial F}{\partial a} = \frac{\partial F}{\partial g} \cdot f_\star \cdot g'_\star.$$

Nous avions vu comment cette formule permettait de calculer le gradient par rapport aux poids.

Preuve. Tout d'abord

$$\frac{\partial F}{\partial a} = \frac{\partial F}{\partial g} \cdot \frac{\partial g}{\partial a}.$$

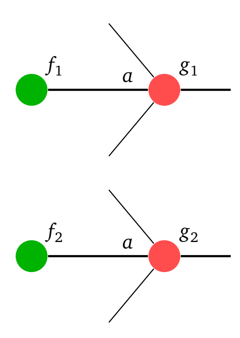
Or

$$\frac{\partial g}{\partial a} = f_\star \cdot g'_\star$$

car $g_\star = g(\cdots + a f_\star + \cdots)$. La formule s'obtient en dérivant $a \mapsto g(\cdots + a f + \cdots)$ par rapport à la variable a.

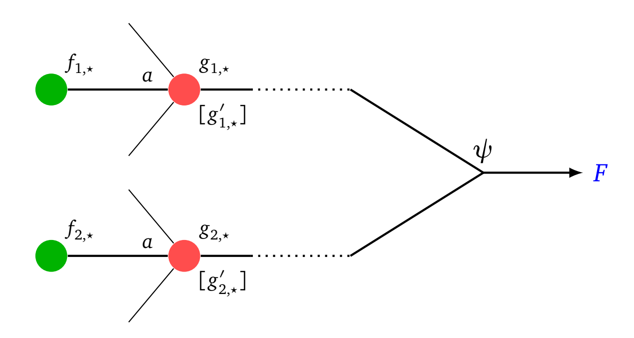
Cas de la convolution.

On se place maintenant dans une couche de convolution. Les neurones de cette couche ont des poids en commun, il faut donc en tenir compte dans les formules. Imaginons une couche de convolution avec deux neurones.



Ces deux neurones ont un poids commun a, par contre les entrées pour ce poids $f_{1,\star}$ et $f_{2,\star}$ peuvent être différentes et les sorties $g_{1,\star}$ et $g_{2,\star}$ également (même si les fonctions d'activation g_1 et g_2 sont les mêmes). Le réseau continue. À la fin, nous obtenons un fonction de sortie F qui dépend des sorties g_1 et g_2 de nos deux neurones :

$$F = \psi(g_1, g_2).$$



Nous avons la nouvelle formule :

$$\frac{\partial F}{\partial a} = \frac{\partial F}{\partial g_1} \cdot f_{1,\star} \cdot g'_{1,\star} + \frac{\partial F}{\partial g_2} \cdot f_{2,\star} \cdot g'_{2,\star}.$$

Preuve. Tout d'abord nous avons $F = \psi(g_1, g_2)$, donc

$$\frac{\partial F}{\partial g_1} = \frac{\partial \psi}{\partial x}(g_1, g_2) \quad \text{et} \quad \frac{\partial F}{\partial g_2} = \frac{\partial \psi}{\partial y}(g_1, g_2).$$

On se souvient de la formule de dérivation de la composition de

$$F(a) = \psi\big(u(a), v(a)\big)$$

qui est

$$F'(a) = u'(a)\frac{\partial \psi}{\partial x}(u(a), v(a)) \ + \ v'(a)\frac{\partial \psi}{\partial y}(u(a), v(a)).$$

Comme on peut aussi écrire que :

$$F = \psi\big(g_1(\cdots + af_1 + \cdots), g_2(\cdots + af_2 + \cdots)\big)$$

alors

$$\begin{aligned}\frac{\partial F}{\partial a} &= f_1 \cdot g_1'(\cdots + af_1 + \cdots)\frac{\partial \psi}{\partial x}\big(g_1(\cdots + af_1 + \cdots), g_2(\cdots + af_2 + \cdots)\big) \\ &+ f_2 \cdot g_2'(\cdots + af_1 + \cdots)\frac{\partial \psi}{\partial y}\big(g_1(\cdots + af_1 + \cdots), g_2(\cdots + af_2 + \cdots)\big).\end{aligned}$$

Ce qui donne bien :

$$\frac{\partial F}{\partial a} = f_{1,\star} \cdot g_{1,\star}' \cdot \frac{\partial F}{\partial g_1} + f_{2,\star} \cdot g_{2,\star}' \cdot \frac{\partial F}{\partial g_2}.$$

Si les fonctions d'activation de la couche de convolution sont l'identité alors $g_{1,\star}' = 1$ et $g_{2,\star}' = 1$, on obtient dans ce cas la formule :

$$\frac{\partial F}{\partial a} = f_{1,\star} \cdot \frac{\partial F}{\partial g_1} + f_{2,\star} \cdot \frac{\partial F}{\partial g_2}.$$

On retient que chaque entrée associée au poids a contribue proportionnellement à sa valeur dans le calcul de la dérivée partielle par rapport à ce poids a.

Plus généralement, pour des entrées f_i, $i = 1, \ldots, n$, pour n neurones d'une couche de convolution ayant pour sortie g_i, $i = 1, \ldots, n$, et pour un poids a partagé par ces neurones :

$$\frac{\partial F}{\partial a} = \sum_{i=1}^{n} f_{i,\star} \cdot g_{i,\star}' \cdot \frac{\partial F}{\partial g_i}.$$

Convolution avec Python

Chapitre 13

Vidéo ■ partie 13. Convolution avec Python

Python permet de calculer facilement les produits de convolution.

1. Convolution et matrices

1.1. Convolution

Le module `scipy` fournit une fonction `convolve2d()` qui calcule le produit de convolution $A \star M$ de deux tableaux *numpy*.

```
import numpy as np
from scipy import signal

A = np.array([[2,1,3,0],[1,1,0,5],[3,3,1,0],[2,0,0,2]])
M = np.array([[1,0,2],[2,1,0],[1,0,3]])

B = signal.convolve2d(A, M, mode='same', boundary='fill')
```

Ce qui correspond à :

$$A = \begin{pmatrix} 2 & 1 & 3 & 0 \\ 1 & 1 & 0 & 5 \\ 3 & 3 & 1 & 0 \\ 2 & 0 & 0 & 2 \end{pmatrix}, \qquad M = \begin{pmatrix} 1 & 0 & 2 \\ 2 & 1 & 0 \\ 1 & 0 & 3 \end{pmatrix} \quad \text{et} \quad B = A \star M = \begin{pmatrix} 5 & 9 & 10 & 0 \\ 7 & 17 & 19 & 16 \\ 10 & 12 & 11 & 0 \\ 5 & 10 & 13 & 5 \end{pmatrix}.$$

L'option `mode='same'` indique que la matrice de sortie B doit avoir la même taille que la matrice d'entrée A. L'option `boundary='fill'` correspond à l'ajout d'une rangée de zéros virtuels sur les bords de A.

C'est un bon exercice de programmer sa propre fonction qui calcule la convolution !

1.2. Variantes

- *Convolution de même taille.* C'est celle que l'on vient de voir et que l'on a définie dans le chapitre précédent. La matrice $B = A \star M$ est de même taille que la matrice A. Pour les calculs, on peut être amené à rajouter des zéros virtuels sur les bords de la matrice A.

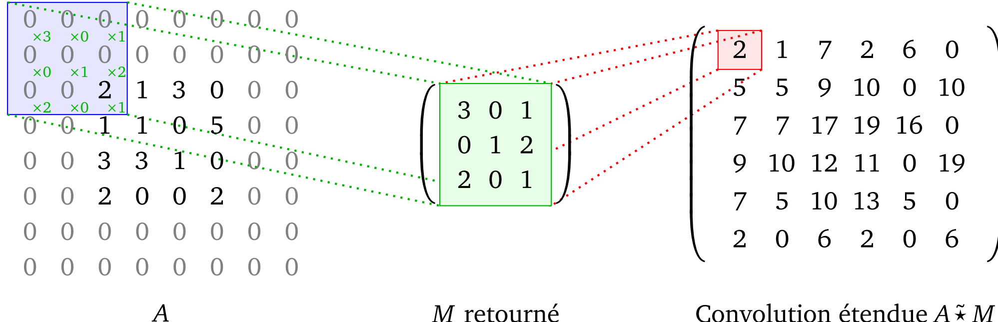
  ```
  B1 = signal.convolve2d(A, M, mode='same', boundary='fill')
  ```

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 3 & 0 & 0 \\ 0 & 1 & 1 & 0 & 5 & 0 \\ 0 & 3 & 3 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix} \qquad \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 2 \\ 2 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 5 & 9 & 10 & 0 \\ 7 & 17 & 19 & 16 \\ 10 & 12 & 11 & 0 \\ 5 & 10 & 13 & 5 \end{pmatrix}$$

A $\quad$ M retourné $\quad$ Convolution de même taille $A \star M$

- *Convolution étendue.* On rajoute deux rangées de zéros virtuels autour de la matrice A. Si A est de taille $n \times p$ et M est de taille 3×3 alors pour cette convolution la matrice B est taille $(n+2) \times (p+2)$.

```
B2 = signal.convolve2d(A, M, mode='full')
```



- *Convolution restreinte.* On ne s'autorise pas à rajouter de zéros virtuels. Pour cette convolution la matrice B est donc de taille $(n-2) \times (p-2)$.

```
B3 = signal.convolve2d(A, M, mode='valid')
```

$$\begin{matrix} 2 & 1 & 3 & 0 \\ 1 & 1 & 0 & 5 \\ 3 & 3 & 1 & 0 \\ 2 & 0 & 0 & 2 \end{matrix} \qquad \begin{pmatrix} 3 & 0 & 1 \\ 0 & 1 & 2 \\ 2 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 17 & 19 \\ 12 & 11 \end{pmatrix}$$

A $\quad$ M retourné $\quad$ Convolution retreinte

Voici les trois matrices : convolution de même taille B_1, convolution étendue B_2, convolution restreinte B_3. La matrice de la convolution étendue B_2, contient la matrice de la convolution classique B_1, qui elle-même contient celle de la convolution restreinte B_3.

$$B_1 = \begin{pmatrix} 5 & 9 & 10 & 0 \\ 7 & 17 & 19 & 16 \\ 10 & 12 & 11 & 0 \\ 5 & 10 & 13 & 5 \end{pmatrix}, \qquad B_2 = \begin{pmatrix} 2 & 1 & 7 & 2 & 6 & 0 \\ 5 & 5 & 9 & 10 & 0 & 10 \\ 7 & 7 & 17 & 19 & 16 & 0 \\ 9 & 10 & 12 & 11 & 0 & 19 \\ 7 & 5 & 10 & 13 & 5 & 0 \\ 2 & 0 & 6 & 2 & 0 & 6 \end{pmatrix}, \qquad B_3 = \begin{pmatrix} 17 & 19 \\ 12 & 11 \end{pmatrix}.$$

1.3. Propriétés mathématiques

La convolution étendue, définie juste au-dessus et que l'on notera $A \tilde{\star} M$ est associative :

$$(A \tilde{\star} M) \tilde{\star} N = A \tilde{\star} (M \tilde{\star} N).$$

Cette associativité signifie que l'on n'a pas besoin d'enchaîner plusieurs convolutions successives. Si on souhaite effectuer la convolution par M puis par N, on peut le faire en une seule fois à l'aide du motif $M \tilde{\star} N$. Pour les réseaux de neurones, il est donc inutile de juxtaposer deux couches de convolution (sans activation) car elles peuvent se réduire à une seule.

Remarque : l'associativité ne serait pas vraie si on ne retournait pas la matrice M dans la définition du calcul de la convolution. C'est donc une des justifications pour laquelle il est nécessaire de retourner M.

1.4. Pooling

Voici une fonction `max_pooling(A,k)` qui à partir d'une matrice A de taille $n \times p$ renvoie la matrice A' de taille $n//k \times p//k$ obtenue par max-pooling.

```
def max_pooling(A,k):
    n, p = A.shape
    B = A.reshape(n//k,k,p//k,k)
    C = B.transpose((0, 2, 1, 3))
    D = C.max(axis=(2,3))
    return D
```

On ne cherche pas expliquer les détails de la fonction. Cependant, on suppose que n et p sont des multiples de k (sinon il faudrait adapter la fonction).
Par exemple, le max-pooling de taille $k = 2$ avec

$$A = \begin{pmatrix} 16 & 23 & 22 & 21 & 12 & 13 \\ 0 & 5 & 18 & 2 & 17 & 15 \\ 10 & 20 & 4 & 9 & 7 & 8 \\ 11 & 14 & 1 & 19 & 6 & 3 \end{pmatrix} \quad \text{donne} \quad A' = \begin{pmatrix} 23 & 22 & 17 \\ 20 & 19 & 8 \end{pmatrix}.$$

Un léger changement renvoie la matrice obtenue par le pooling en moyenne.

```
def average_pooling(A,k):
    n, p = A.shape
    B = A.reshape(n//k,k,p//k,k)
    C = B.transpose((0, 2, 1, 3))
    D = C.mean(axis=(2,3))
    return D
```

Appliquée à la matrice A et $k = 2$, la fonction ci-dessus retourne :

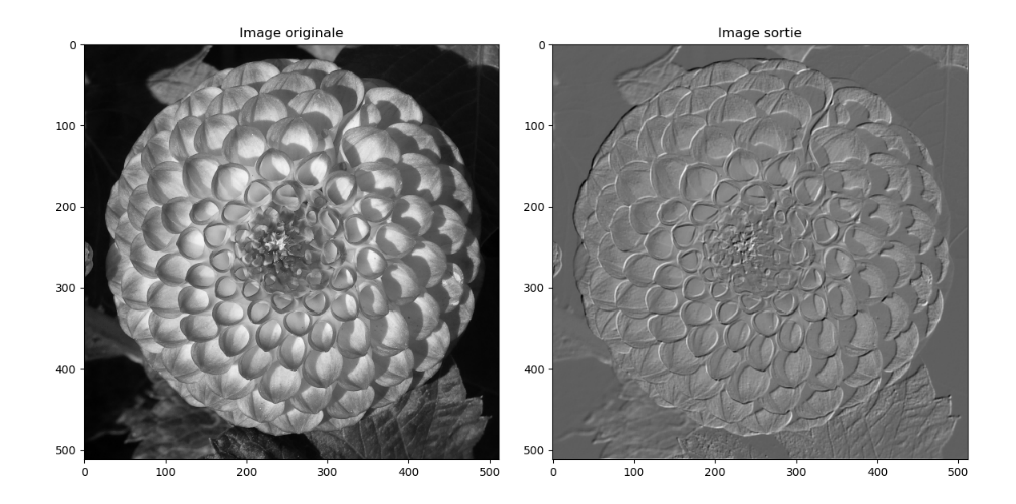
$$A'' = \begin{pmatrix} 11 & 15.75 & 14.25 \\ 13.75 & 8.25 & 6 \end{pmatrix}.$$

2. Traitement d'images

Comme nous l'avons vu dans le chapitre précédent, il s'agit juste d'appliquer une convolution. Une image en niveau de gris est considérée comme une matrice A dont les coefficients sont des entiers compris entre 0 et 255. Le motif utilisé dans l'illustration ci-dessous est donné par la matrice :

$$M = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

L'image de sortie correspond à la matrice B obtenue par la convolution $B = A \star M$. Ici le motif M permet l'estampage (mise en relief).




```
import numpy as np
import matplotlib.pyplot as plt

# Partie A - Importer une image comme un tableau
import imageio
A = imageio.imread('image_avant.png')

# Partie B - Motif de convolution
M = np.array([[-2,-1,0],[-1,1,1],[0,1,2]])

# Partie C - Calcul de la convolution
from scipy import signal
B = signal.convolve2d(A, M, boundary='fill', mode='same')

# Partie D - Affichage des images avant/après
fig = plt.figure(figsize = (10,5))

ax = plt.subplot(1,2,1)
ax.set_title("Image originale")
ax.imshow(A, cmap='gray')

bx = plt.subplot(1,2,2)
bx.set_title("Image sortie")
bx.imshow(B, cmap='gray')

plt.show()

# Partie E - Sauvegarde de l'image
B = np.clip(B,0,255)    # limite les valeurs entre 0 et 255
B = B.astype(np.uint8)  # conversion en entiers
imageio.imwrite('image_apres.png', B)
```

Explications.

- On utilise le module `imageio` (qui n'est pas installé par défaut) et permet de transformer une image en un tableau *numpy* (`imread()`) ou l'opération inverse (`imwrite()`).
- On prend soin de rendre les coefficients de la matrice B entiers entre 0 et 255.

- Il existe de multiples variantes du format d'image « .png », aussi vos images peuvent s'afficher différemment selon votre visionneuse d'images !

On renvoie au chapitre « Convolution » pour d'autres exemples de motifs intéressants.

3. Convolution : une dimension

3.1. Convolution et vecteurs

Le module *numpy* permet la convolution de vecteurs.

```
f = np.array([1,0,3,5,1])
g = np.array([1,2,3])
h = np.convolve(f,g,'same')
```

$$f \star g = (1,0,3,5,1) \star (1,2,3) = (2,6,11,20,17).$$

La convolution étendue $f \tilde{\star} g$ s'obtient à l'aide du paramètre `'full'`. On renvoie à la dernière section du chapitre « Convolution : une dimension » :

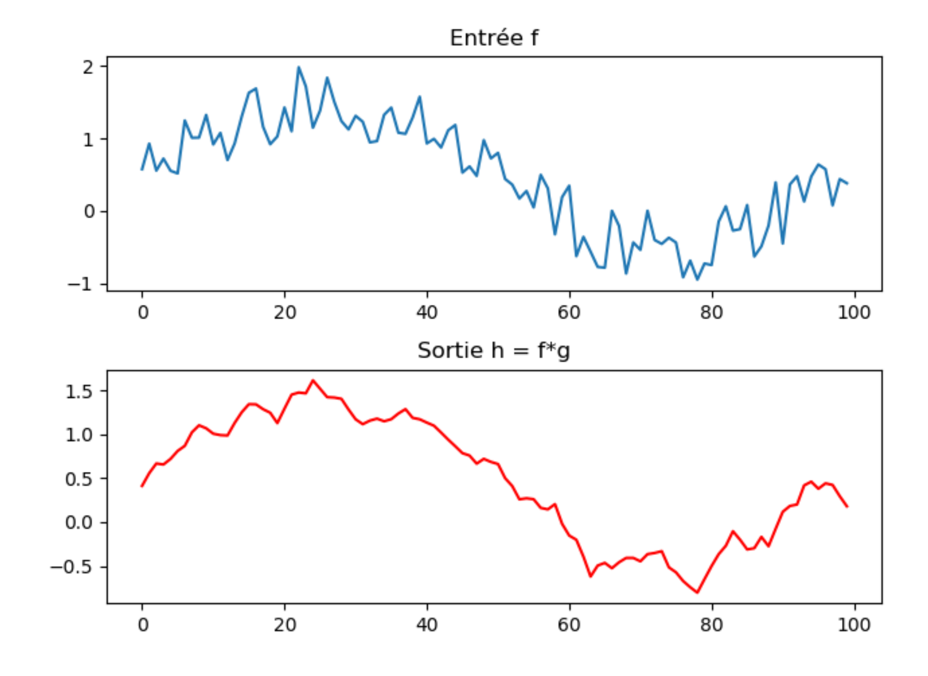
```
h = np.convolve(f,g,'full')
```

donne :

$$h = f \tilde{\star} g = (1,0,3,5,1) \tilde{\star} (1,2,3) = (1,2,6,11,20,17,3).$$

3.2. Visualisation

La visualisation permet de mieux comprendre la convolution en une dimension. Ici un vecteur f est obtenu en ajoutant un bruit aléatoire à une sinusoïde. Le vecteur g est un motif qui effectue une moyenne mobile (ici de taille 5). La convolution $h = f \star g$ correspond à un lissage du tracé de f.




```
N = 100
f = np.sin(np.linspace(0,2*np.pi,N)) + np.random.random(N)
g = 1/5*np.array([1,1,1,1,1])
```

```
h = np.convolve(f,g,'same')

ax = plt.subplot(2,1,1)
ax.set_title("Entrée f")
plt.plot(f)

ax = plt.subplot(2,1,2)
ax.set_title("Sortie h = f*g")
plt.plot(h,color='red')

plt.show()
```

3.3. Convolution et polynômes

La convolution est une opération que vous pratiquez déjà sans le savoir ! Si on code un polynôme

$$P(X) = a_n X^n + a_{n-1} X^{n-1} + a_1 X + a_0$$

par la liste de ses coefficients :

$$[P] = (a_n, a_{n-1}, \ldots, a_1, a_0),$$

alors la multiplication $P \times Q$ de deux polynômes correspond au produit de convolution des listes des coefficients :

$$[P \times Q] = [P] \mathbin{\widetilde{\star}} [Q].$$

Par exemple, pour

$$P = X^3 + 2X^2 + 3X + 4 \qquad [P] = (1, 2, 3, 4)$$

et

$$Q = 5X^2 + 6X + 7 \qquad [Q] = (5, 6, 7),$$

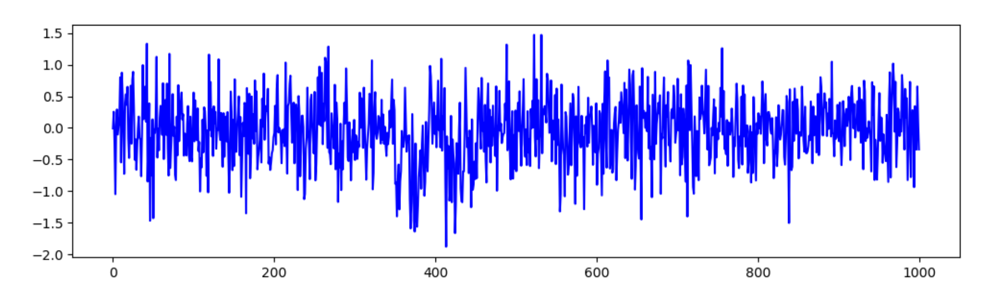
on calcule la convolution étendue des coefficients :

$$[P] \mathbin{\widetilde{\star}} [Q] = (1, 2, 3, 4) \mathbin{\widetilde{\star}} (5, 6, 7) = (5, 16, 34, 52, 45, 28).$$

Cela correspond aux coefficients de $P \times Q$:

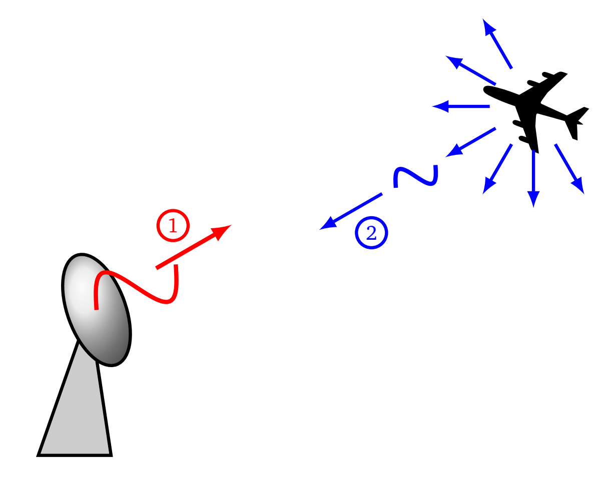
$$P \times Q = 5X^5 + 16X^4 + 34X^3 + 52X^2 + 45X + 28.$$

3.4. Traitement du signal

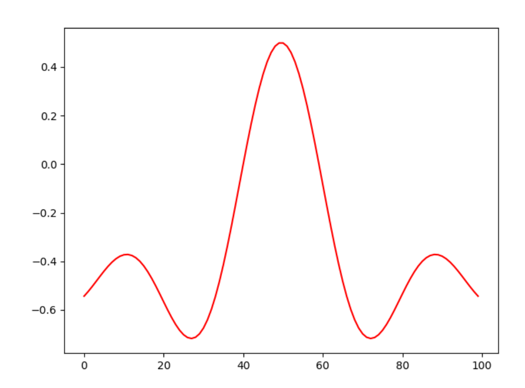


Signal reçu par un radar : où est l'avion ?

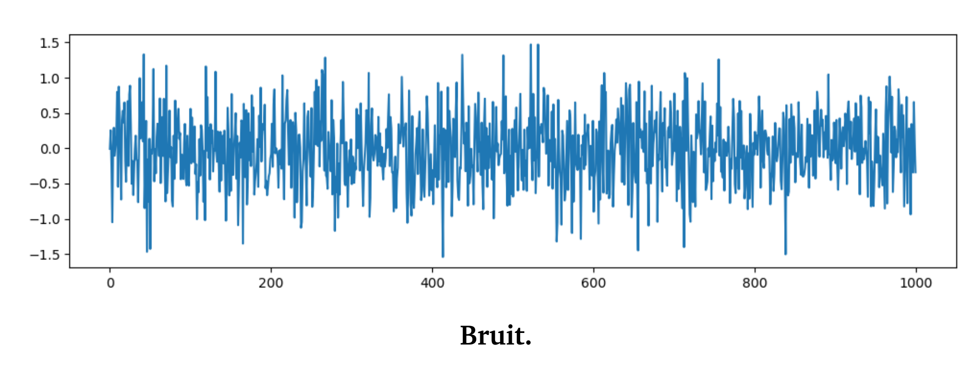
Radar. Nous avons vu que la convolution permet le traitement d'un signal : lissage d'une courbe (voir ci-dessus), mais aussi d'autres opérations (voir le chapitre « Convolution : une dimension »). Une autre application est la détection radar. En voici le principe : le radar émet un signal, les ondes atteignent un objet (un avion par exemple) et sont renvoyées par réflexion, le radar reçoit ce signal réfléchi. Le temps mis par les ondes pour effectuer l'aller-retour permet de calculer la distance à l'objet.



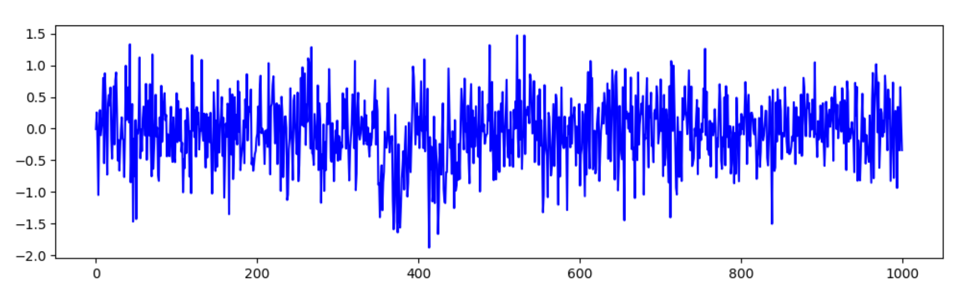
Dans la pratique c'est plus compliqué : le signal émis est bien connu, mais le signal reçu est composé de l'onde réfléchie à laquelle se superpose un bruit aléatoire lié à l'environnement (les nuages, les oiseaux...).



Impulsion g émise par le radar.



Bruit.

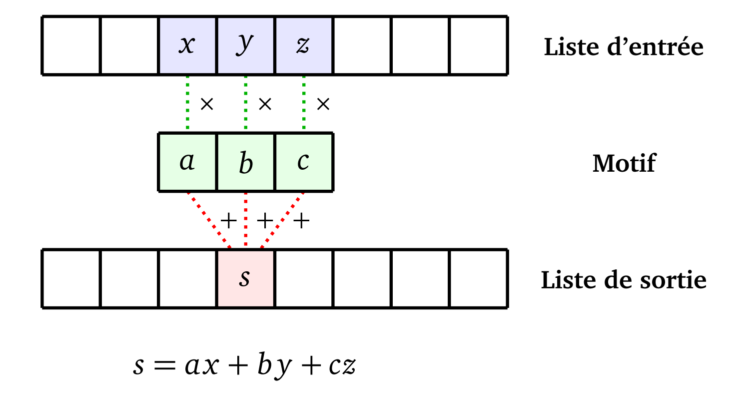


Signal f reçu par le radar

Sur l'exemple ci-dessus, le signal f est obtenu comme la somme d'un bruit (un signal aléatoire) et une copie de l'impulsion g entre les abscisses 350 et 450. Regardez bien la différence entre le bruit et le signal f autour de l'abscisse 400 !

Comment retrouver la position de l'impulsion g à partir du signal reçu f ?

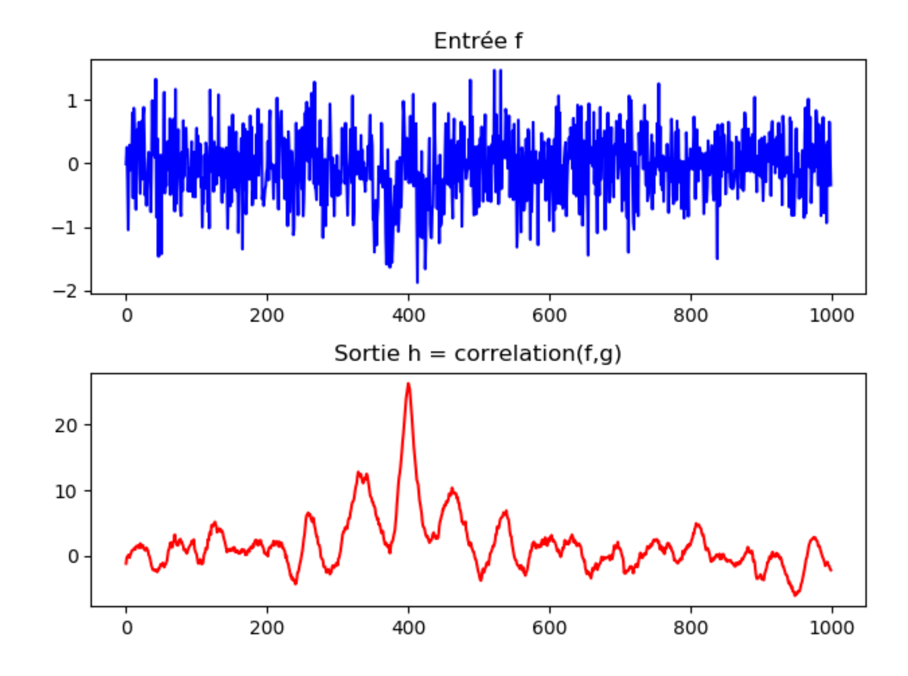
Corrélation.
Nous n'allons pas utiliser la convolution, mais la ***corrélation*** qui est une opération similaire sauf que le motif n'est pas renversé.



$$s = ax + by + cz$$

Par exemple si $f = (1, 0, 3, 5, 1)$ et $g = (1, 2, 3)$ alors la corrélation de f avec g vaut $h = (2, 10, 21, 16, 7)$.

Détection.
On note g le signal émis par le radar, c'est un vecteur (ici de longueur 100). On note f le signal reçu par le radar, c'est un vecteur (ici de longueur 1000). On calcule la corrélation h de f avec g, c'est un vecteur (ici de longueur 1000).



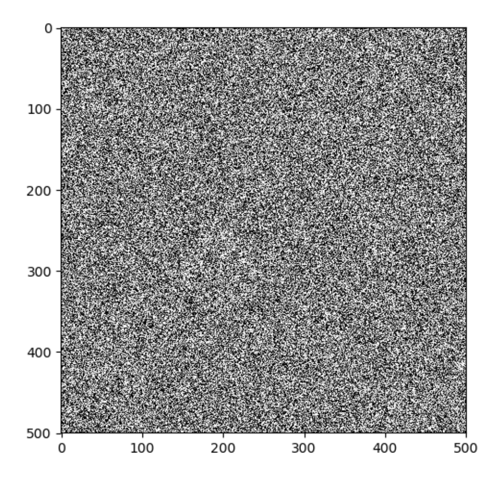
Le pic observé correspond au signal renvoyé par l'avion. Sur l'exemple, ce signal est détecté en position 400 (400 millisecondes par exemple). Ce qui permet de calculer la distance du radar à l'avion. Plusieurs mesures permettraient même de déterminer sa vitesse.

4. Corrélation en deux dimensions

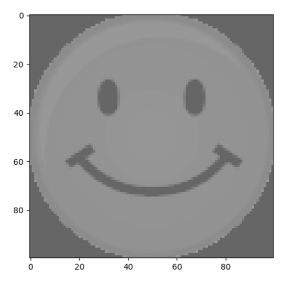
Sur cette photo vous reconnaissez immédiatement un chat !



Mais pour un ordinateur ce n'est qu'un amas de pixels plus ou moins gris.
Ci-dessous, voici une autre image qui montre mieux ce que peut représenter une image pour un ordinateur. Que voyez-vous sur cette image : un chat, un chien ou un bonhomme qui sourit ?



Nous allons voir que la convolution permet de retrouver un motif dans une image, exactement comme on l'a fait avec le radar dans le cas d'une dimension. Voici le motif qu'il fallait « voir » dans l'image ci-dessus.



Tout d'abord, quelques informations pour obtenir une image brouillée. On part d'une image 500 × 500 de pixels (ici en niveaux de gris) aléatoires (image de gauche ci-dessous). On superpose le motif « smiley » de taille 100 × 100 à une partie de cette l'image. On obtient donc une image avec un motif caché (image de droite ci-dessous).

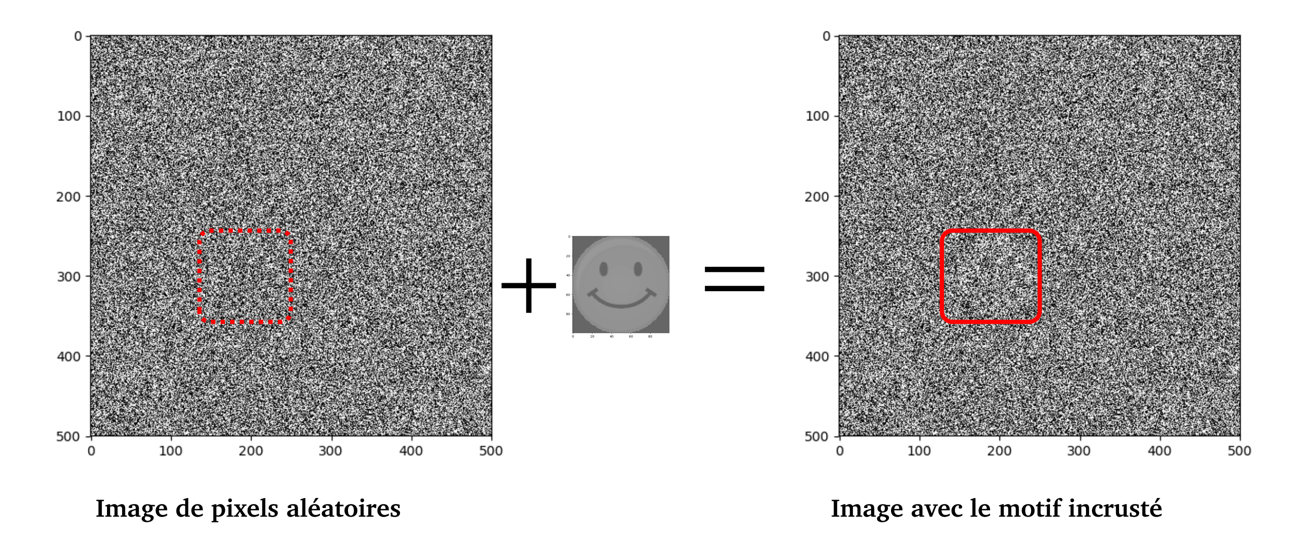
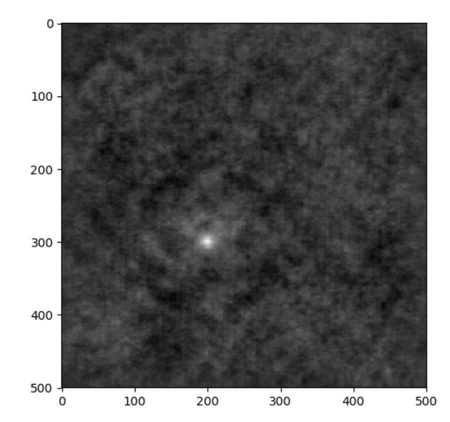


Image de pixels aléatoires **Image avec le motif incrusté**

On calcule la corrélation entre l'image et le motif. La corrélation en deux dimensions, c'est comme la convolution mais sans retourner la matrice du motif. La corrélation indique si le motif est présent ou pas dans l'image et si oui, à quel endroit.



Sur notre exemple la corrélation indique une correspondance autour du point $(200,300)$.

Convolution avec tensorflow/keras

Chapitre 14

Vidéo ■ partie 14.1. Reconnaissance de chiffres
Vidéo ■ partie 14.2. Reconnaissance d'images
Vidéo ■ partie 14.3. Reconnaissance d'images - Chat vs chien
Vidéo ■ partie 14.4. Que voit un réseau de neurones ?

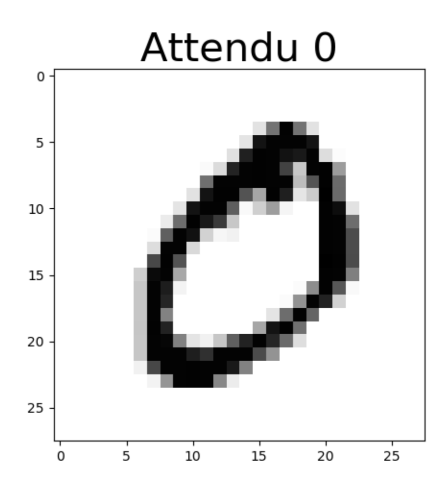
Nous mettons en œuvre ce qui a été vu dans les chapitres précédents au sujet des couches de convolution afin de créer des réseaux de neurones beaucoup plus performants.

1. Reconnaissance de chiffres

On souhaite reconnaître des chiffres écrits à la main.

1.1. Données

Les données sont celles de la base MNIST déjà rencontrée dans le chapitre « Python : tensorflow avec keras - partie 2 ». On rappelle brièvement que cette base contient 60 000 données d'apprentissage (et 10 000 données de test) sous la forme d'images 28×28 en niveau de gris, étiquetées par le chiffre attendu.

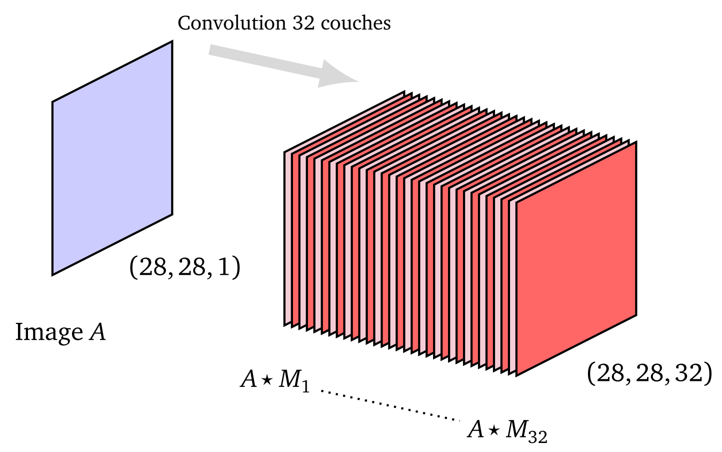


Pour le besoin de la convolution, on conserve chaque image sous la forme d'un tableau 28×28 ; afin de préciser que l'image est en niveau de gris, la taille du tableau *numpy* est en fait $(28, 28, 1)$ (à comparer à la représentation d'une image couleur de taille $(28, 28, 3)$).

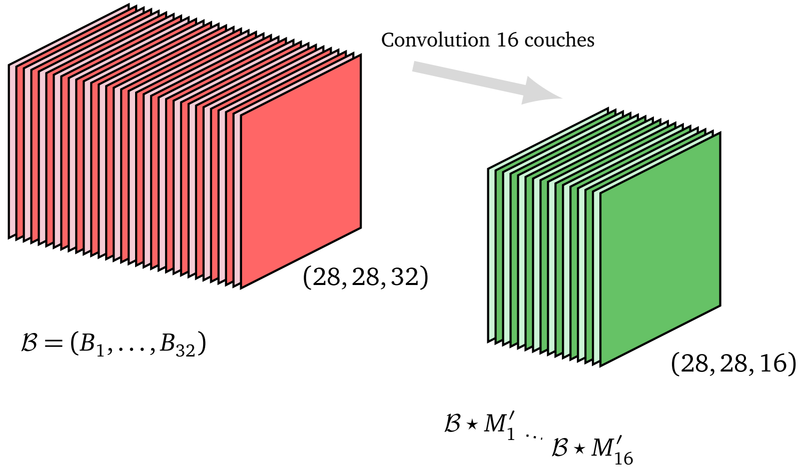
1.2. Modèle

On crée un réseau composé de la façon suivante :

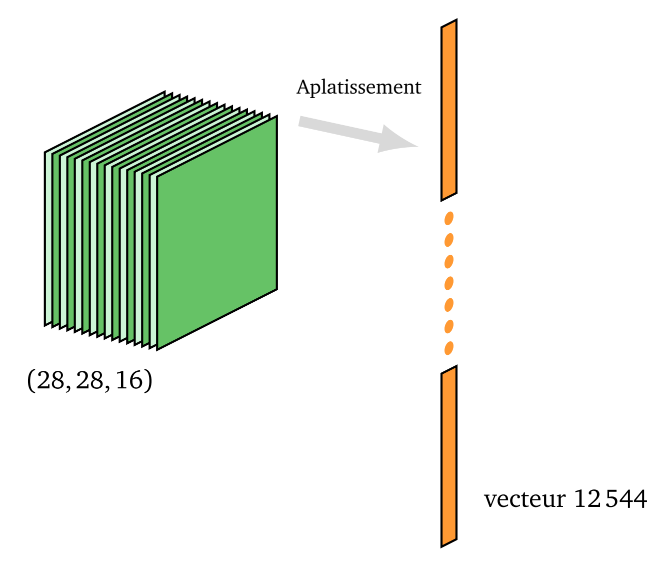
- **Première couche de convolution.** Une couche formée de 32 sous-couches de convolution. Si on note A l'image de départ, chaque sous-couche est la convolution $A \star M_i$ de A par un motif M_i de taille 3×3 ($i = 1, \ldots, 32$). Une entrée de taille $(28, 28, 1)$ est transformée en une sortie de taille $(28, 28, 32)$.
  Chaque sous-couche correspond à un neurone de convolution avec 3×3 arêtes plus un biais, c'est-à-dire 10 poids par sous-couche. Avec nos 32 sous-couches, cela fait au total 320 poids pour cette première couche.



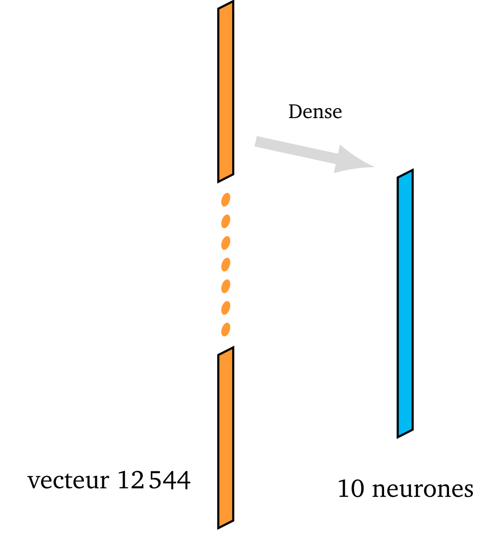
- **Seconde couche de convolution.** Une couche formée de 16 sous-couches de convolution. Si on note $\mathcal{B} = (B_1, \ldots, B_{32})$ les sorties de la couche précédente, qui deviennent maintenant les entrées, alors chaque sous-couche de sortie est une convolution $\mathcal{B} \star M'_i$ par un motif M'_i de taille $3 \times 3 \times 32$. Une entrée de taille $(28, 28, 32)$ est transformée en une sortie de taille $(28, 28, 16)$.
  Chaque sous-couche correspond à un neurone de convolution avec $3 \times 3 \times 32$ arêtes plus un biais, soit 289 poids par sous-couche. Avec nos 16 sous-couches, cela fait un total de 4624 poids pour cette seconde couche.



- **Aplatissement.** Chaque sortie de taille $(28, 28, 16)$ est reformatée en un (grand) vecteur de taille 12 544 ($= 28 \times 28 \times 16$). Il n'y a aucun poids associé à cette transformation : ce n'est pas une opération mathématique, c'est juste un changement du format des données.

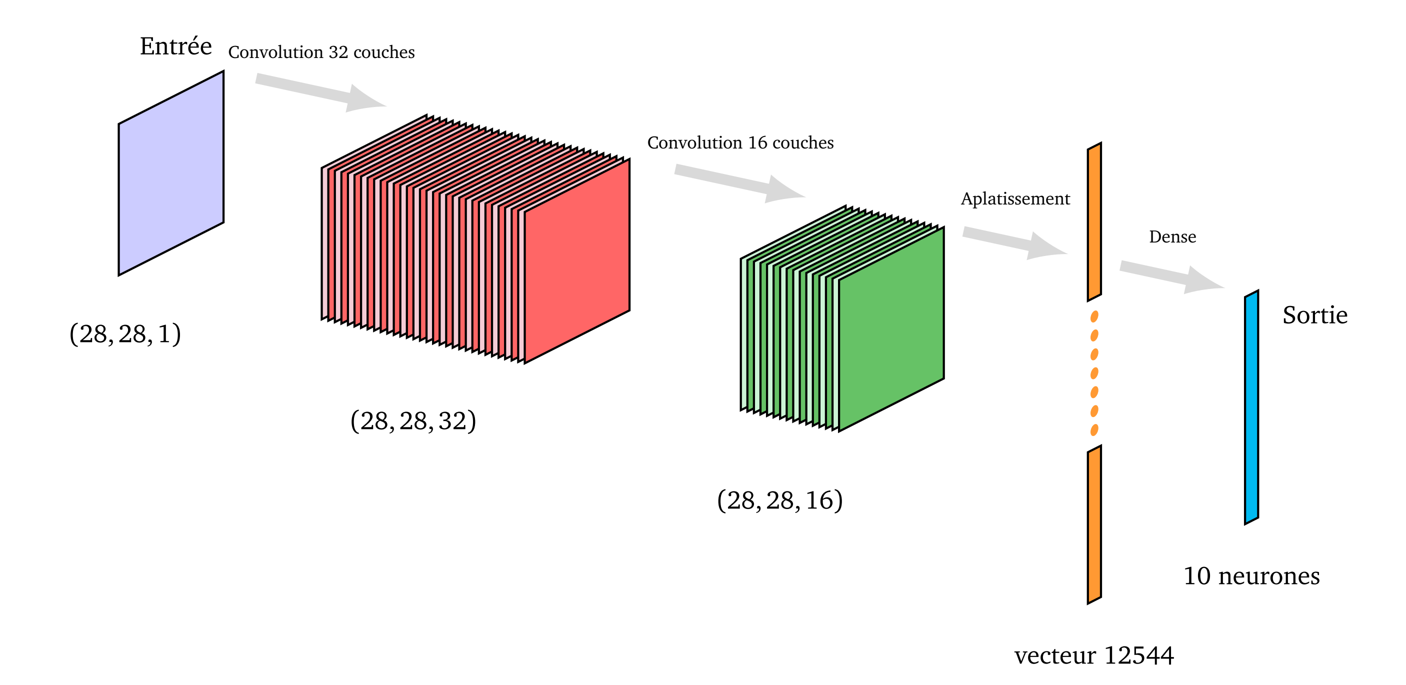


- **Couche dense.** C'est une couche « à l'ancienne » composée de 10 neurones, chaque neurone étant relié aux 12 544 entrées. En tenant compte d'un biais par neurone cela fait 125 450 $(= (12\,544 + 1) \times 10)$ poids pour cette couche.

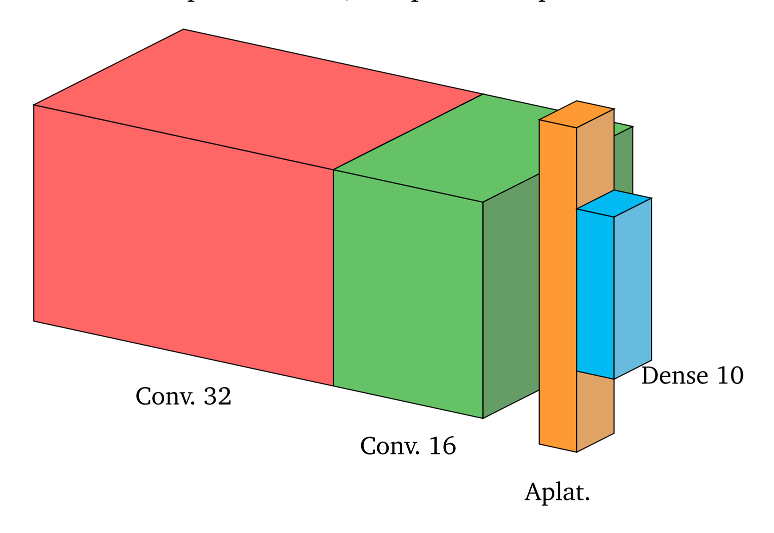


- **Bilan.** Il y a en tout 130 394 poids à calculer.

Voici l'architecture complète du réseau.



On résume l'architecture du réseau par des blocs, chaque bloc représentant une transformation.



1.3. Programme

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten

### Partie A - Création des données
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(X_train_data,Y_train_data),(X_test_data,Y_test_data) = mnist.load_data()

N = X_train_data.shape[0]  # 60 000 données
```

```
X_train = np.reshape(X_train_data, (N,28,28,1))
X_test = np.reshape(X_test_data, (X_test_data.shape[0],28,28,1))

X_train = X_train/255  # normalisation
X_test = X_test/255

Y_train = to_categorical(Y_train_data, num_classes=10)
Y_test = to_categorical(Y_test_data, num_classes=10)

### Partie B - Réseau de neurones
modele = Sequential()

# Première couche de convolution : 32 neurones, motif 3x3, activ. relu
modele.add(Conv2D(32, kernel_size=3, padding='same', activation='relu',
                      input_shape=(28,28,1)))

# Deuxième couche de convolution : 16 neurones
modele.add(Conv2D(16, kernel_size=3, padding='same', activation='relu'))

# Aplatissage
modele.add(Flatten())

# Couche de sortie : 10 neurones
modele.add(Dense(10, activation='softmax'))

# Descente de gradient
modele.compile(loss='categorical_crossentropy',
               optimizer='adam',
               metrics=['accuracy'])

print(modele.summary())

# Calcul des poids
modele.fit(X_train, Y_train, batch_size=32, epochs=5)

### Partie C - Résultats
score = modele.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

1.4. Explications

Une couche de convolution est ajoutée avec *tensorflow/keras* par une commande du type :

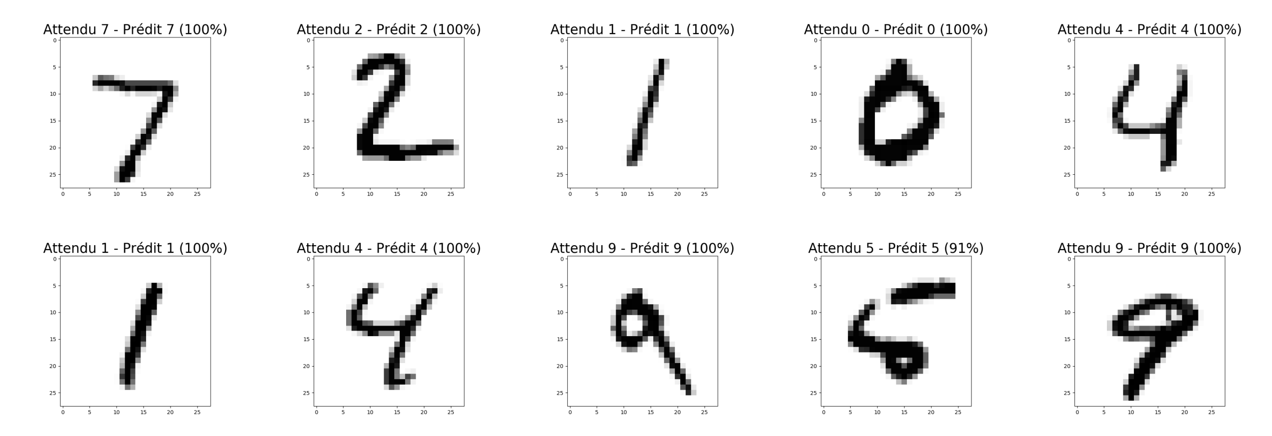
```
modele.add(Conv2D(16, kernel_size=3, padding='same', activation='relu'))
```

qui correspond à une couche composée de 16 sous-couches de convolution. Chacune de ces sous-couches correspond à une convolution par un motif de taille 3×3 (option `kernel_size=3`) et conserve la taille de l'image (option `padding='same'`). La fonction d'activation pour chaque neurone est « ReLU ».

Note : en fait le calcul d'une « convolution » avec *tensorflow*/*keras* est un calcul de corrélation, c'est-à-dire que les coefficients du motif ne sont pas retournés. Cela n'a pas vraiment d'importance car ces coefficients sont déterminés par rétropropagation et cela reste transparent à l'usage.

1.5. Résultats

Après 5 époques, on obtient une précision de 99% sur les données de test, ce qui est un gros progrès par rapport au 95% obtenus dans le chapitre « Python : tensorflow avec keras - partie 2 ». Il faut bien avoir conscience que les derniers pourcentages de précision sont de plus en plus difficiles à acquérir !



Sur les exemples ci-dessus toutes les prédictions sont correctes avec en plus une quasi-certitude de 100%, à l'exception de l'avant-dernière image qui conduit à une bonne prédiction mais avec une certitude plus faible. En effet le vecteur de sortie pour cette image est :

$$Y = (0, 0, 0, 0, 0, 0.911, 0.089, 0, 0, 0).$$

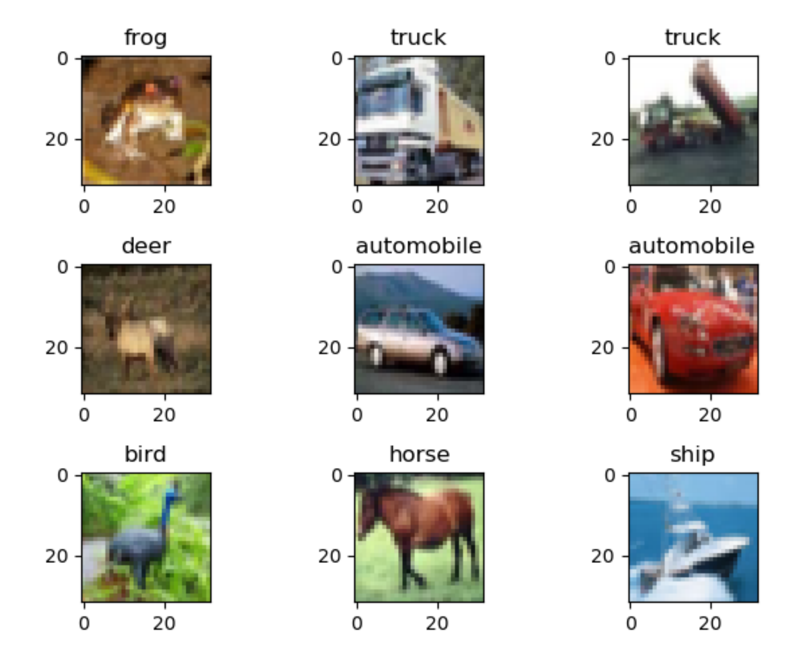
Le chiffre 5 est donc prédit à 91% avec un léger doute avec le chiffre 6 à 9%, ce qui est tout à fait rassurant vu que ce chiffre est très mal dessiné !

2. Reconnaissance d'images

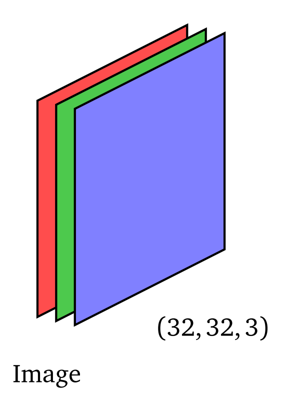
On souhaite reconnaître des objets ou des animaux sur de petites photos.

2.1. Données

On rappelle que la base CIFAR-10, déjà rencontrée dans le chapitre « Python : tensorflow avec keras - partie 2 » contient 50 000 images d'apprentissage, réparties en 10 catégories.

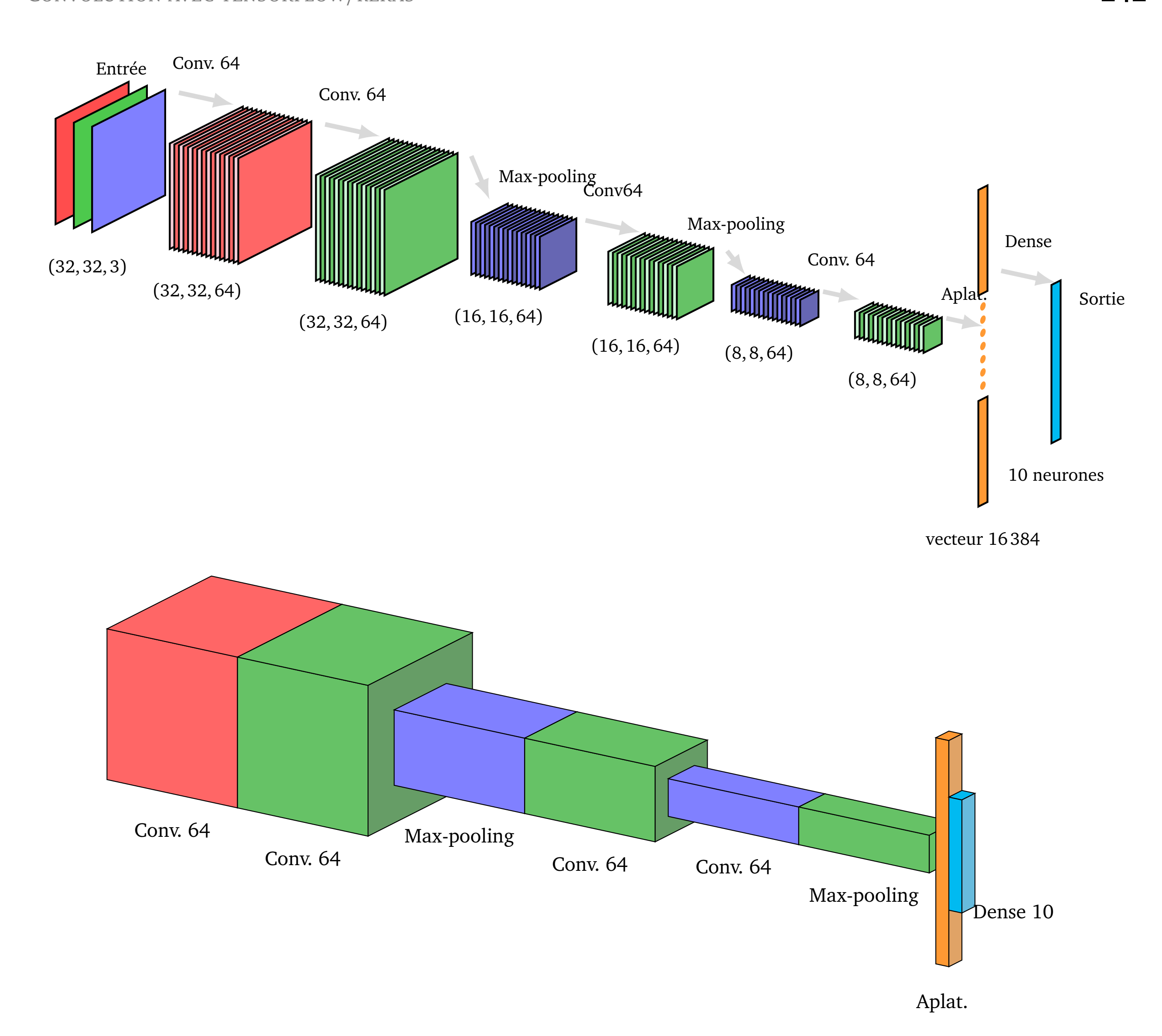


Chaque image possède 32×32 pixels de couleurs. Une entrée correspond donc à un tableau $(32, 32, 3)$.



2.2. Modèle

L'architecture du réseau utilisé est composée de plusieurs couches de convolution. Pour diminuer le nombre de poids à calculer, on intercale des couches de pooling (regroupement de termes). Ces pooling sont des max-pooling de taille 2×2. De tels regroupements divisent par 4 la taille des données en conservant les principales caractéristiques, ce qui fait que la couche de neurones suivante possèdera 4 fois moins de poids à calculer.



Il y a en tout 153 546 poids à calculer. Sans les deux couches de pooling, il y aurait 767 946 poids.

2.3. Programme

```
modele = Sequential()

# Première couche de convolution : 64 neurones, motif 3x3, activ. relu
modele.add(Conv2D(64, kernel_size=3, padding='same', activation='relu',
                  input_shape=(32,32,3)))

# Deuxième couche de convolution : 64 neurones
modele.add(Conv2D(64, kernel_size=3, padding='same', activation='relu'))

# Mise en commun (pooling)
modele.add(MaxPooling2D(pool_size=(2, 2)))

# Troisième couche de convolution : 64 neurones
modele.add(Conv2D(64, kernel_size=3, padding='same', activation='relu'))

# Mise en commun (pooling)
```

```
modele.add(MaxPooling2D(pool_size=(2, 2)))

# Quatrième couche de convolution : 64 neurones
modele.add(Conv2D(64, kernel_size=3, padding='same', activation='relu'))

# Aplatissage
modele.add(Flatten())

# Couche de sortie : 10 neurones
modele.add(Dense(10, activation='softmax'))
```

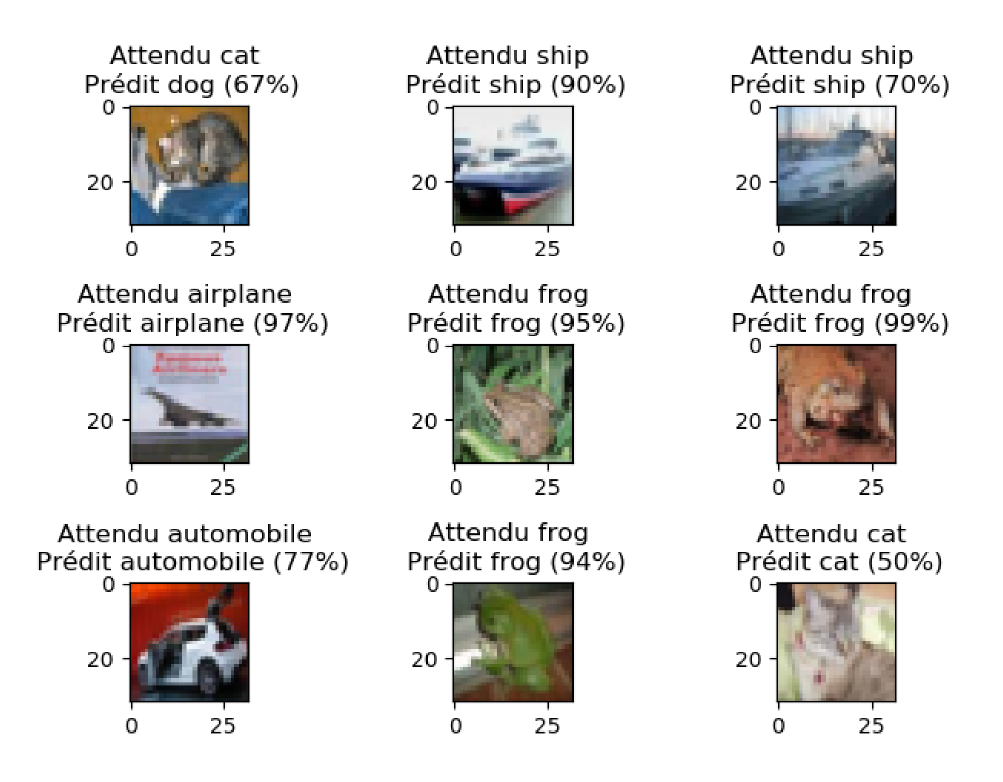
2.4. Explications

La nouveauté est ici l'ajout de couches de pooling par la commande :

```
modele.add(MaxPooling2D(pool_size=(2, 2)))
```

2.5. Résultats

Après une dizaine d'époques, on obtient plus de 80% de précision sur les données d'entraînement et un peu moins de 75% sur les données de test. Ainsi les trois quarts des images sont correctement prédites.



3. Chat vs chien

Il s'agit de décider si une photo est celle d'un chat ou d'un chien.

3.1. Données

Nous allons traiter une situation plus réaliste que celles vues jusqu'ici, car les données sont de « vraies » photos qu'il faut préparer avant de commencer les calculs.

Récupérer les données.

Télécharger les données sur le site « Kaggle » qui a organisé un concours de reconnaissance de photos :

https://www.kaggle.com/c/dogs-vs-cats/data

Le fichier à récupérer est « train.zip » (600 Mo).

Les photos. Le fichier contient 25 000 photos couleurs de différentes tailles :

- 12 500 photos de chats, nommées sous la forme `cat.1234.jpg`,
- 12 500 photos de chiens, nommées sous la forme `dog.1234.jpg`.





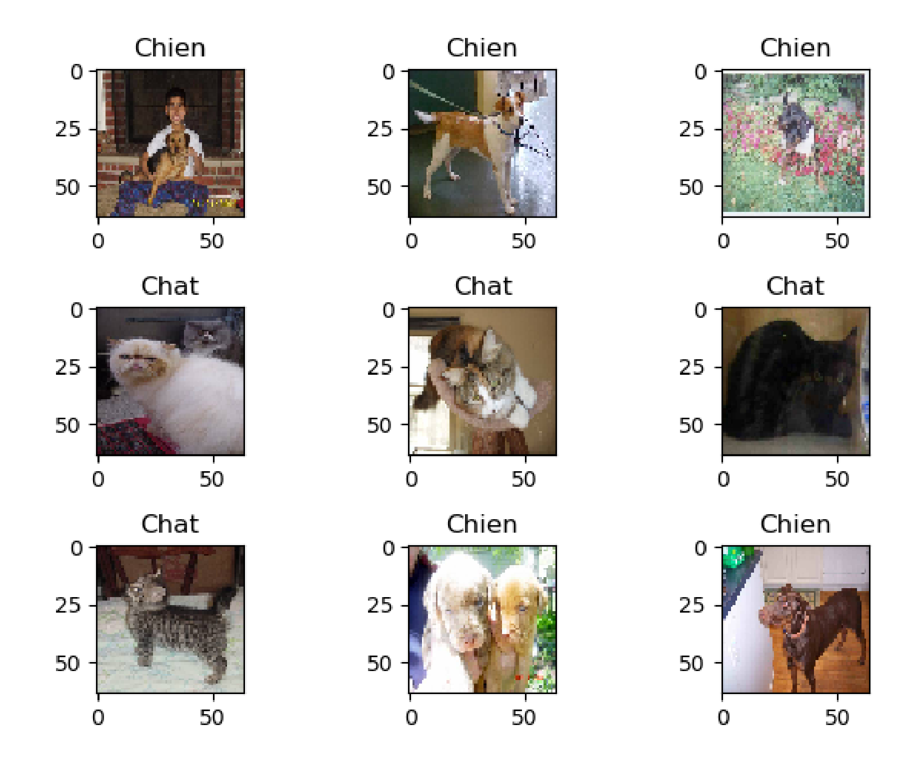
Organiser les photos. Répartir à la main les photos dans des sous-répertoires avec la structure suivante :

```
donnees/
    train/
        cats/
        dogs/
    test/
        cats/
        dogs/
```

Déplacer 10 000 photos de chats dans `donnees/train/cats` puis 10 000 photos de chiens dans `donnees/train/dogs`. Déplacer les 2 500 photos de chats restantes dans `donnees/test/cats` puis les 2 500 photos de chiens restantes dans `donnees/test/dogs`.

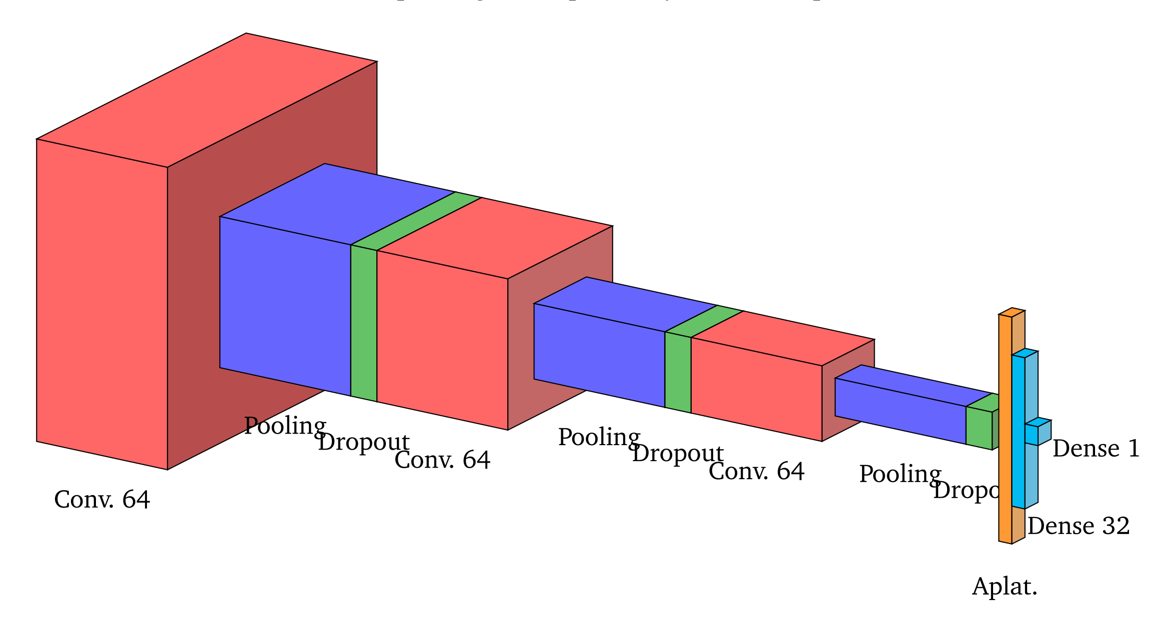
Note : vous pouvez commencer avec seulement 1 000 photos par sous-répertoire pour des calculs plus rapides.

Redimensionnement des photos. Les photos vont toutes être redimensionnées à la même taille, par exemple 64×64. Ces tâches préliminaires seront effectuées pour nous par *keras*. De plus, comme les photos représentent une grande quantité de données, nous allons voir comment *keras* permet de ne mettre en mémoire qu'une petite quantité d'images à chaque étape de la descente de gradient.



3.2. Modèle

On alterne : couche de convolution, pooling et dropout. Il y a 206 685 poids à calculer.



3.3. Programme

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Conv2D,Flatten,MaxPooling2D,Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Partie A. Données
```

```
# A télécharger sur https://www.kaggle.com/c/dogs-vs-cats/data
train_directory = 'monrepertoire/donnees/train'
test_directory  = 'monrepertoire/donnees/test'
image_width = 64
image_height = 64
nb_train_images = 20000

# Transforme les images en données d'apprentissage :
# (a) reformate les images en taille unique,
# (b) créé une classification (à partir de chaque sous-répertoire)
# 0 pour les chats  et 1 pour les chiens
train_datagen = ImageDataGenerator(rescale =1./255)
training_set = train_datagen.flow_from_directory(train_directory,
                                       target_size=(image_width,image_height),
                                       batch_size= 32,
                                       shuffle=True, seed=13,
                                       class_mode='binary')

# Partie B. Réseau
modele = Sequential()

modele.add(Conv2D(64, kernel_size=3, padding='same', activation='relu',
                      input_shape=(image_width,image_height,3)))
modele.add(MaxPooling2D(pool_size=(2, 2)))
modele.add(Dropout(0.5))

modele.add(Conv2D(64, kernel_size=3, padding='same', activation='relu'))
modele.add(MaxPooling2D(pool_size=(2, 2)))
modele.add(Dropout(0.5))

modele.add(Conv2D(64, kernel_size=3, padding='same', activation='relu'))
modele.add(MaxPooling2D(pool_size=(2, 2)))
modele.add(Dropout(0.5))

modele.add(Flatten())
modele.add(Dense(32, activation='relu'))
modele.add(Dense(1, activation='sigmoid'))

modele.compile(loss='binary_crossentropy',
               optimizer='adam',
               metrics=['accuracy'])

# Partie C. Apprentissage
history = modele.fit_generator(training_set,
                        steps_per_epoch = nb_train_images // 32,
                        epochs = 10)
```

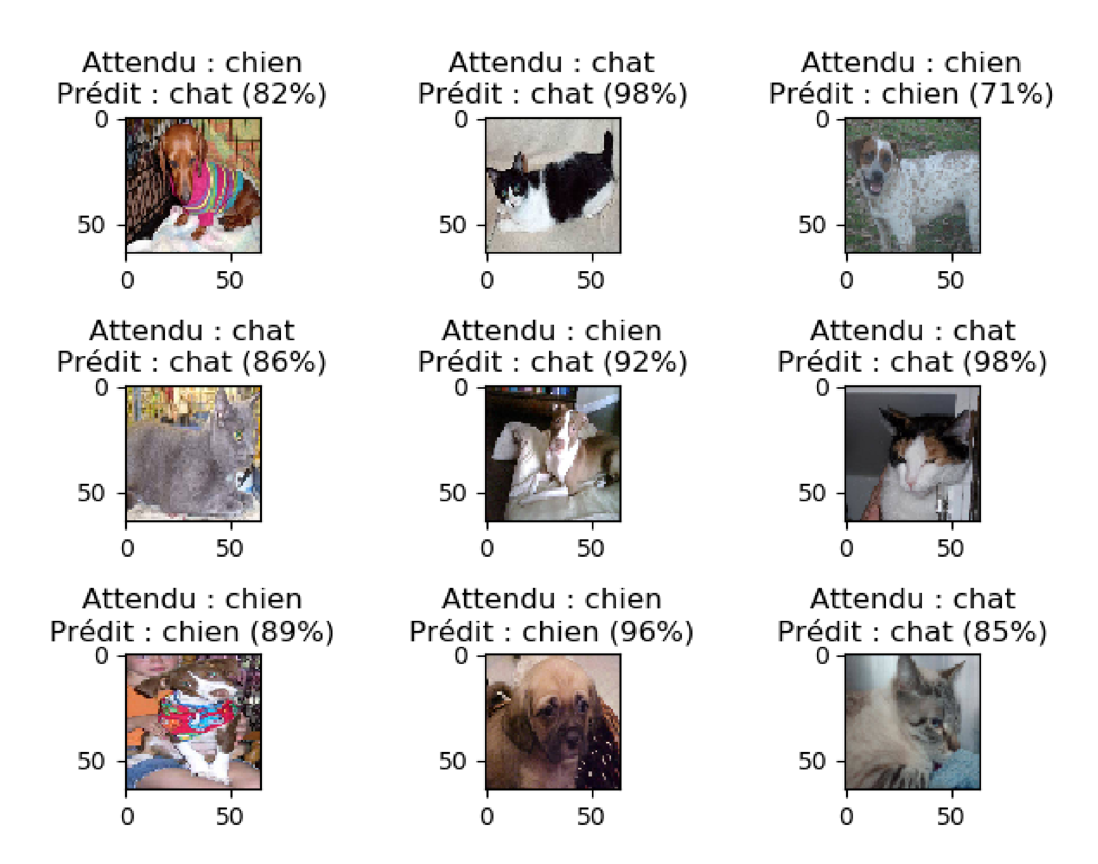
3.4. Explications

La nouveauté est ici l'ajout de couches de dropout par la commande :

```
modele.add(Dropout(0.5))
```

3.5. Résultats

Après pas mal de calculs et une quarantaine d'époques, on obtient une précision de 85%, ce qui est tout fait correct, même si on est loin des 99% de précision des meilleurs résultats du concours « Kaggle ». À vous de jouer !



On montre ci-dessus des images de test avec deux photos mal identifiées, peut-être à cause d'éléments extérieurs (le gilet coloré en haut à gauche, la serviette au milieu).

4. Que voit un réseau de neurones ?

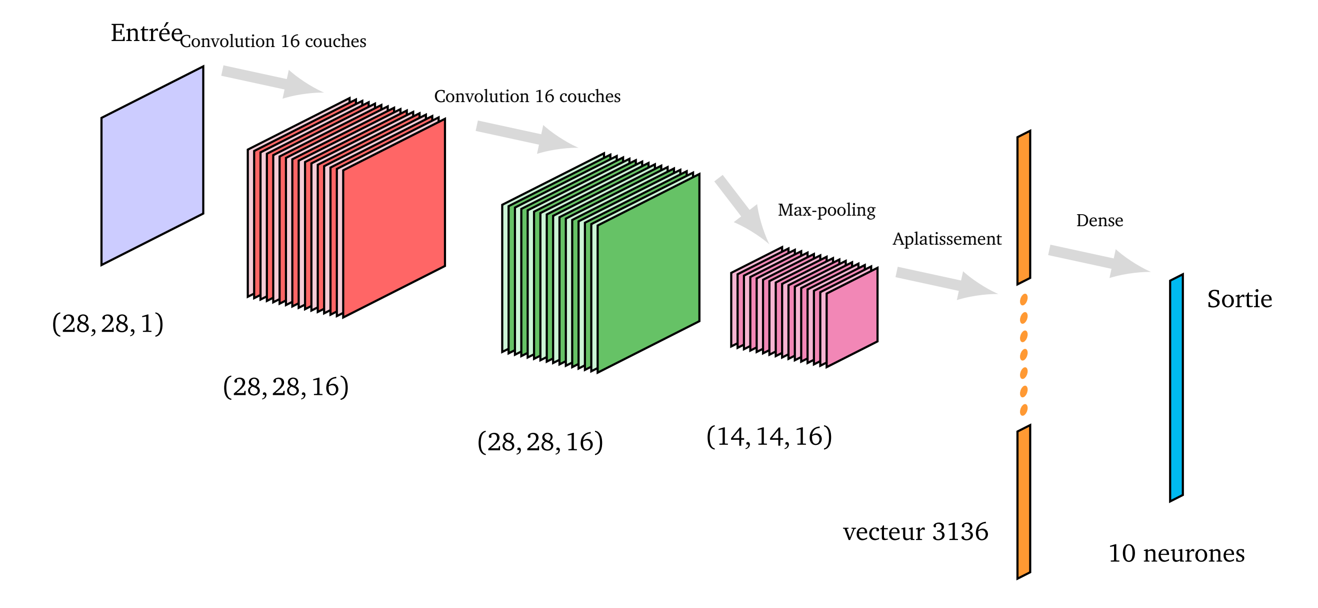
Une des difficultés des réseaux de neurones est qu'ils agissent comme une « boîte noire ». Une fois bien paramétrés, ils répondent efficacement à un problème. Mais même avec plus 99% de bonnes décisions, l'utilisateur peut avoir un doute sur la réponse donnée par une machine (en particulier pour une question médicale, la conduite autonome...). Il est donc intéressant d'inspecter le fonctionnement interne d'un réseau déjà paramétré en essayant de comprendre ce que font les couches intermédiaires.

4.1. Un exemple

On reprend l'exemple de la reconnaissance de chiffres manuscrits de la base MNIST. On construit un réseau simple :

- **Entrée.** Une image en niveau de gris de taille 28×28.

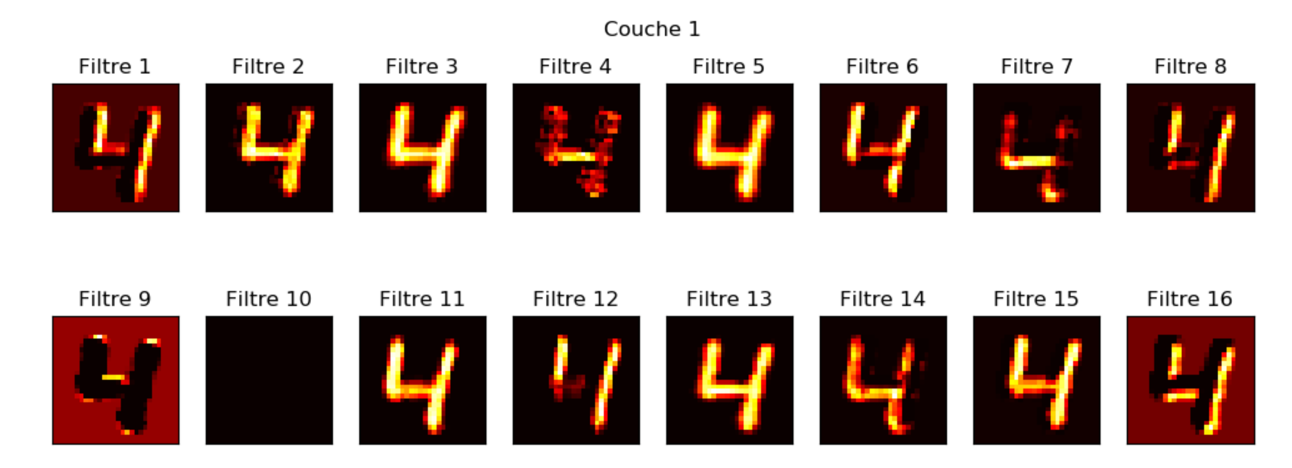
- **Couche 1.** Une couche de convolution avec 16 filtres (autrement dit 16 sous-couches).
- **Couche 2.** Une autre couche de convolution avec encore 16 filtres.
- **Couche 3.** Une couche de max-pooling 2 × 2.
- **Aplatissement.**
- **Couche dense et sortie.** Une couche dense de 10 neurones qui est aussi la couche de sortie.



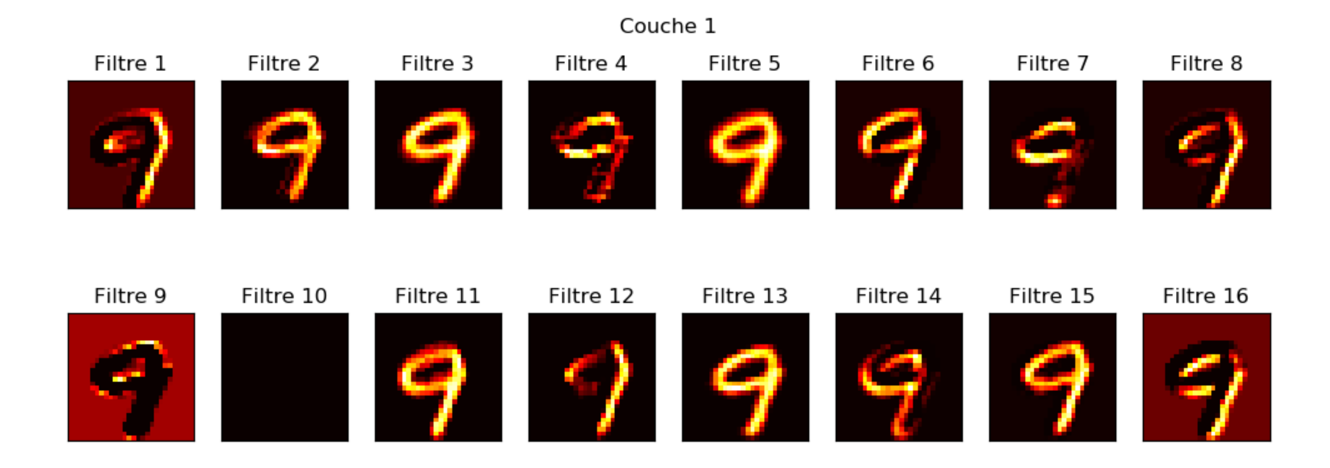
On entraîne le réseau sur plusieurs époques jusqu'à ce qu'il prédise correctement 99% des images de test. Nous allons visualiser l'action des couches 1, 2 et 3.

4.2. Visualisation de couches

Première couche. Voici ce que « voit » le réseau à la sortie de la couche 1, tout d'abord avec un exemple où l'entrée est un chiffre 4.



La couleur d'un pixel représente la valeur de la sortie d'un neurone de convolution (blanc/élevée, noir/faible). Comme il y a 16 sous-couches cela donne 16 images différentes pour le même chiffre 4.
Les filtres ont différentes actions et certaines peuvent être déterminées. Par exemple, il semble que les filtres 1 et 8 mettent en avant les lignes verticales, alors que les filtres 4 et 7 renforcent les lignes horizontales. Le filtre 16 effectue une sorte de mise en relief. D'autres sont difficiles à interpréter.
Voici l'action de la même couche 1 sur un autre chiffre.



Deuxième couche. Voyons maintenant ce qu'il se passe à la sortie de la couche 2, pour notre chiffre 4 et notre chiffre 9. Attention les filtres sont de nouveau numérotés de 1 à 16 mais n'ont rien à voir avec les filtres de la couche 1.

Couche 2

Filtre 1 Filtre 2 Filtre 3 Filtre 4 Filtre 5 Filtre 6 Filtre 7 Filtre 8

Filtre 9 Filtre 10 Filtre 11 Filtre 12 Filtre 13 Filtre 14 Filtre 15 Filtre 16

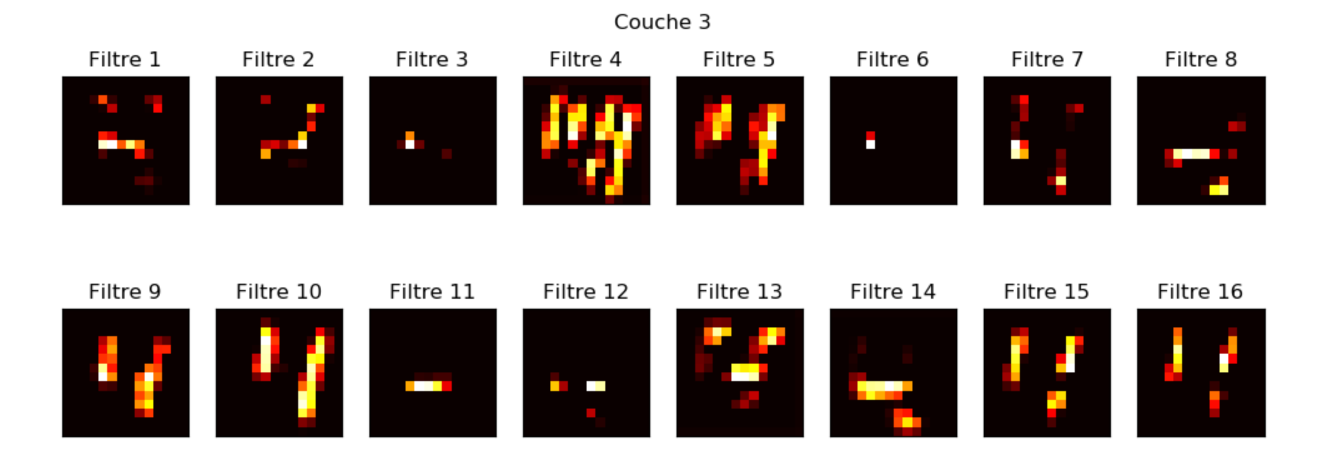
Couche 2

Filtre 1 Filtre 2 Filtre 3 Filtre 4 Filtre 5 Filtre 6 Filtre 7 Filtre 8

Filtre 9 Filtre 10 Filtre 11 Filtre 12 Filtre 13 Filtre 14 Filtre 15 Filtre 16

La deuxième couche renvoie une vision plus abstraite de nos chiffres et les distingue. Par exemple le filtre 11 différencie le chiffre 4 (un seul trait horizontal) du chiffre 9 (deux traits horizontaux), de même avec le filtre 10 et ses traits verticaux.

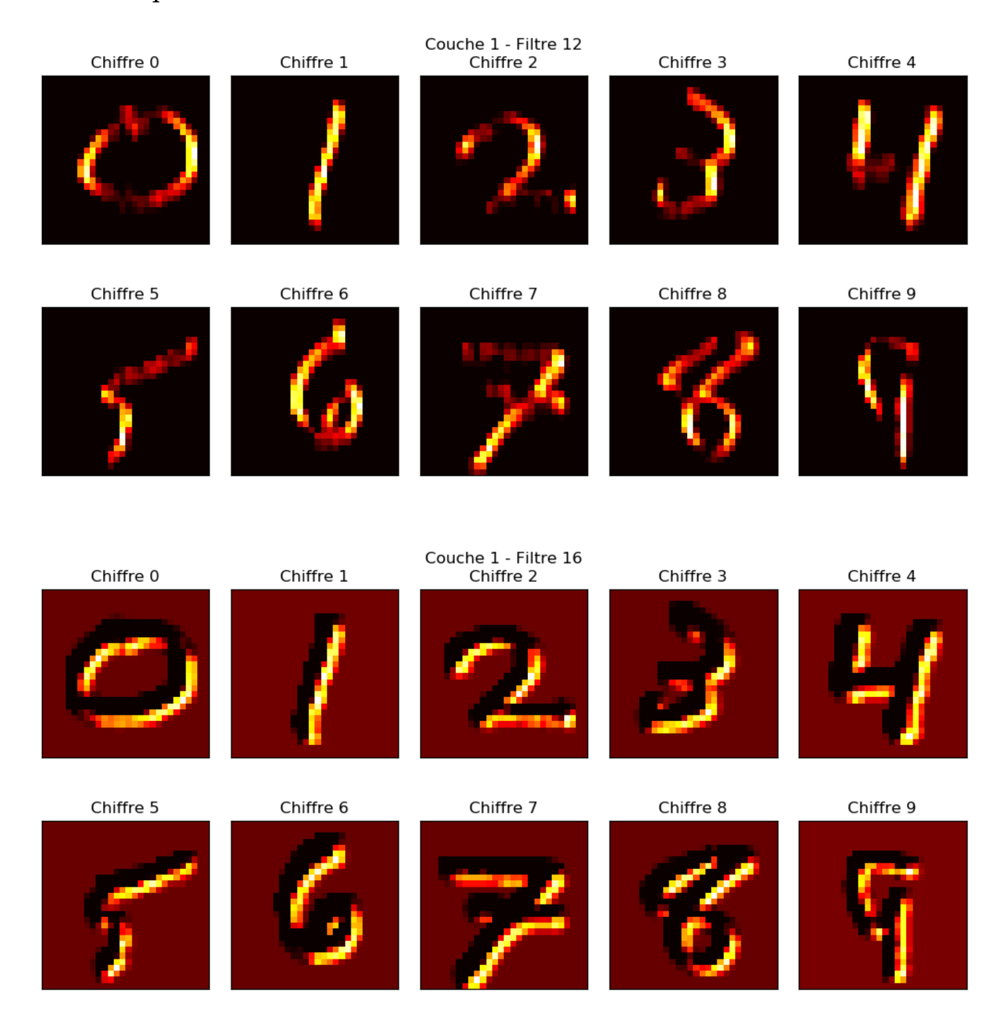
Troisième couche. C'est une couche de max-pooling. Cette fois les filtres de la couche 3 correspondent aux filtres le couche 2. Même si les tailles sont réduites d'un facteur 2 × 2, on voit sur l'exemple du chiffre 4 ci-dessous que les principales caractéristiques sont conservées.



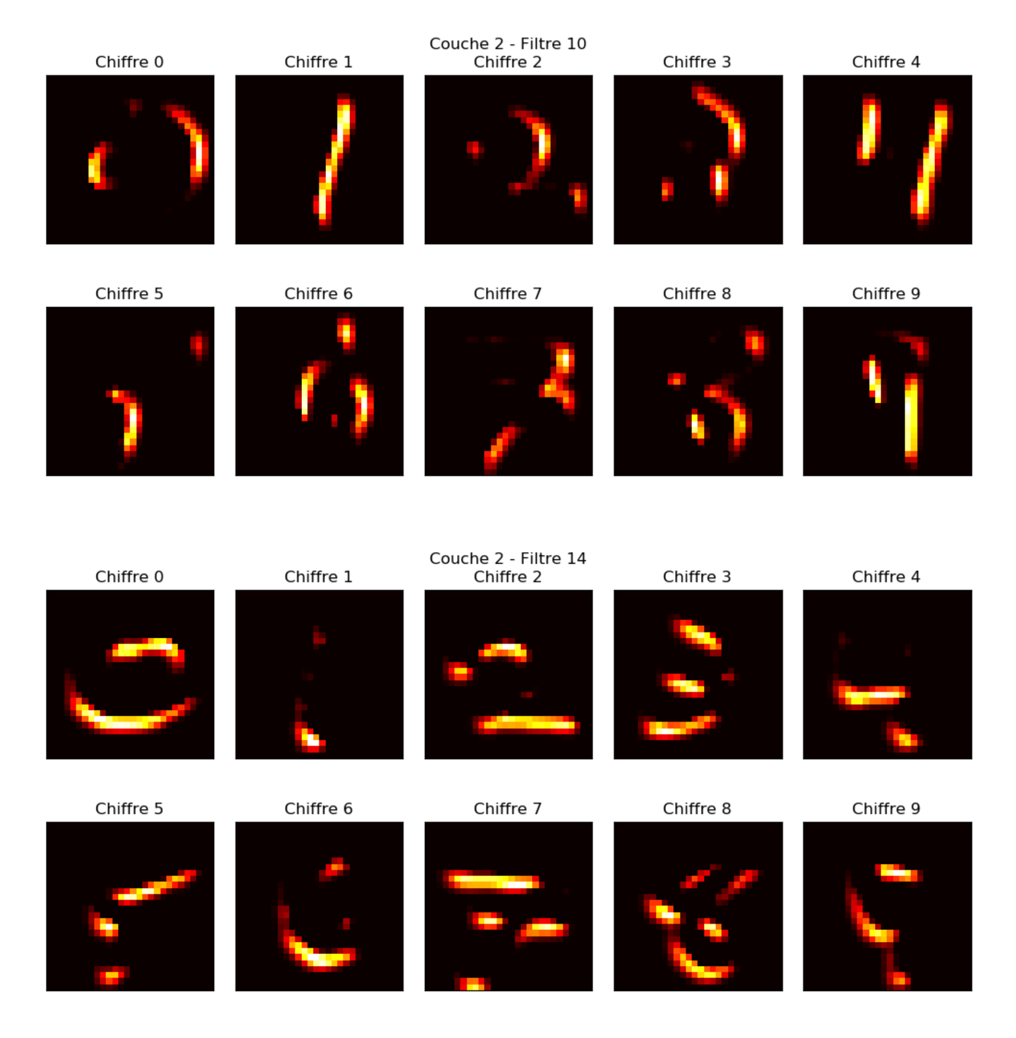
4.3. Visualisation de filtres

Un autre point de vue est de se concentrer sur un seul filtre d'une seule couche et de regarder comment celui-ci traite différentes images.

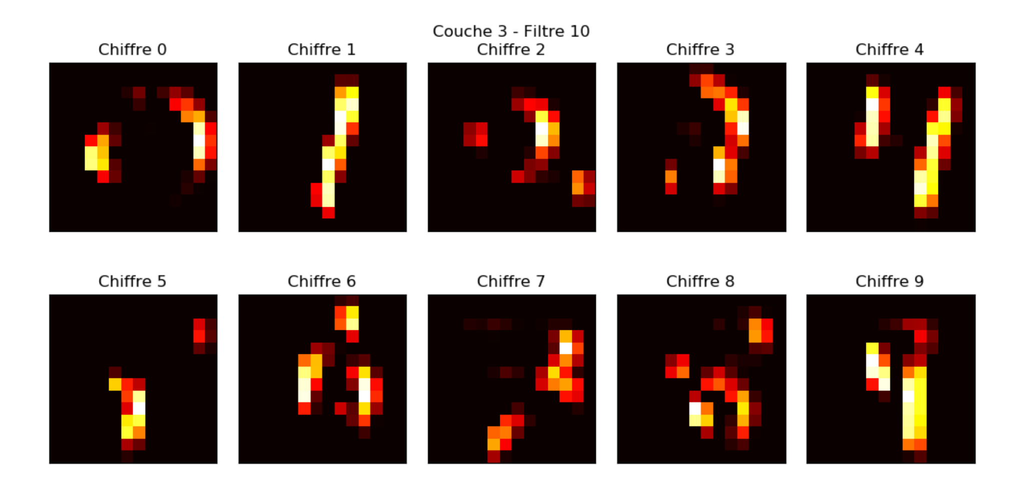
Première couche. Voici deux exemples pour la couche 1 : le filtre 12 qui met en évidence les segments verticaux et le filtre 16 qui met en relief.

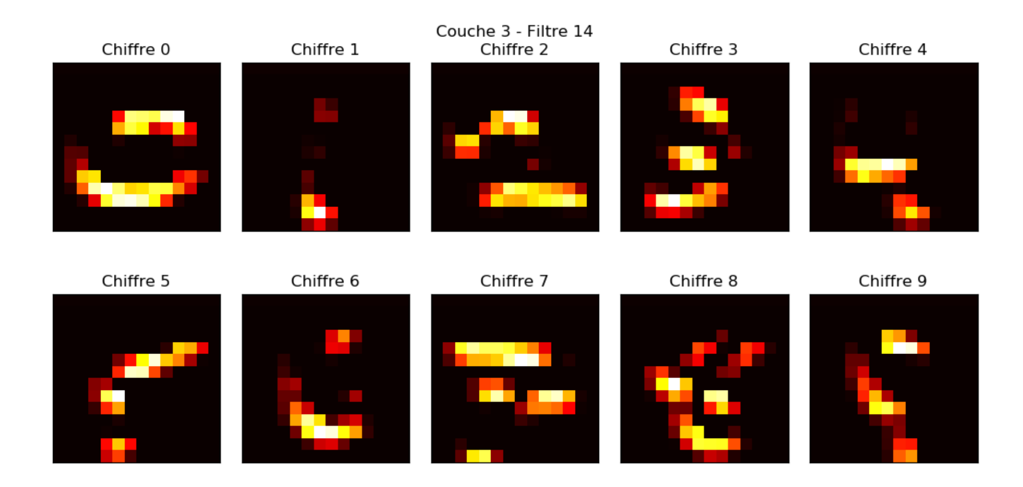


Deuxième couche. Le filtre 10 renforce les lignes verticales. Le filtre 14 renforce les lignes horizontales, on voit ainsi qu'il permet de détecter le chiffre 1, c'est le seul chiffre qui n'active (presque) aucun pixel pour ce filtre. (On rappelle que les filtres de cette deuxième couche n'ont rien à voir avec les filtres de la première couche.)



Troisième couche. La troisième couche est une couche de pooling, ainsi ses filtres correspondent aux filtres de la couche précédente.




Couche 3 - Filtre 14
Chiffre 0
Chiffre 1
Chiffre 2
Chiffre 3
Chiffre 4
Chiffre 5
Chiffre 6
Chiffre 7
Chiffre 8
Chiffre 9

Chapitre
15

Tenseurs

Vidéo ■ partie 15. Tenseurs

Un tenseur est un tableau à plusieurs dimensions, qui généralise la notion de matrice et de vecteur et permet de faire les calculs dans les réseaux de neurones.

1. Tenseurs (avec *numpy*)

1.1. Qu'est ce qu'un tenseur ?

Un tenseur est un tableau de nombres à plusieurs dimensions. Voici des exemples de tenseurs :

- un vecteur V est un tenseur de dimension 1 (c'est un tableau à une seule dimension),
- une matrice M est un tenseur de dimension 2 (c'est un tableau à deux dimensions),
- un 3-tenseur T est un tableau à 3 dimensions.

Voici un exemple pour chacune des situations avec leur définition *numpy*.

$$V = \begin{pmatrix} 5 \\ 7 \\ 8 \end{pmatrix}$$

```
V = np.array([5, 7, 8])
```

$$M = \begin{pmatrix} 6 & 0 & 2 \\ 3 & 2 & 4 \end{pmatrix}$$

```
M = np.array([[6, 0, 2],
              [3, 2, 4]])
```

```
T = np.array([ [[5, 1, 0],
                [3, 3, 2]],
               [[1, 1, 0],
                [5, 1, 7]],
               [[0, 0, 1],
                [8, 1, 9]] ])
```

On accède aux éléments en utilisant les coordonnées :

- $V[i]$,
- $M[i, j]$ (ou bien $M[i][j]$),
- $T[i, j, k]$ (ou bien $T[i][j][k]$).

Plus généralement un ***n*-tenseur** T est un tableau de nombres ayant n dimensions. On accède à ses éléments en précisant ses n coordonnées $T[i_1, i_2, \ldots, i_n]$.

Il n'est pas très difficile de définir des n-tenseurs, ce sont juste des tableaux de nombres. Par contre, il est plus difficile de visualiser un tenseur à partir de la dimension 4. Une matrice peut être vue comme une liste de vecteurs. De même, un 3-tenseur est une liste de matrices, un 4-tenseur une liste de 3-tenseurs...

Une autre façon de voir un 4-tenseur est de le présenter comme une matrice dans laquelle chaque élément est en fait une matrice :

$$T = \begin{pmatrix} M_{1,1} & M_{1,2} & \cdots \\ M_{2,1} & M_{2,2} & \cdots \\ \vdots & \vdots & \vdots \end{pmatrix}.$$

Remarque.

- La notion de tenseur avec *tensorflow* est beaucoup plus sophistiquée que celle de tenseur avec *numpy*. Nous en aurons un aperçu dans la section suivante.
- Il existe une définition plus générale (et beaucoup plus compliquée) de tenseur en mathématiques. Pour ceux qui connaissent les espaces vectoriels, la différence entre un tenseur mathématique et un tenseur comme présenté ici est la même qu'entre une application linéaire et une matrice.

1.2. Vocabulaire

Il est important de maîtriser le vocabulaire lié aux tenseurs et de ne pas confondre les notions.

- ***Dimension***.
  - — C'est le nombre d'indices qu'il faut utiliser pour accéder à un élément. Chaque indice correspond à un *axe*.
  - — Un vecteur est de dimension 1, une matrice de dimension 2 (le premier axe correspond par exemple aux lignes et le second axe aux colonnes), un n-tenseur est de dimension n car on accède à un élément $T[i_1, i_2, \ldots, i_n]$ en utilisant n indices.
  - — Le nom anglais est *rank* (qui n'a rien à voir avec le rang d'une matrice).
  - — La commande est `T.ndim`.
- ***Taille***.
  - — C'est la liste des longueurs de chaque axe.
  - — Un vecteur de longueur 3 a pour taille (3) que l'on note plutôt $(3,)$ (pour signifier que c'est bien une liste). Une matrice 3×4 a pour taille $(3, 4)$. Un tenseur de taille $(3, 5, 7, 11)$ est un tenseur de dimension 4, le premier indice i_1 varie de 1 à 3 (ou plutôt de 0 à 2 avec *Python*), le second indice i_2 varie de 1 à 5. . .
  - — Le nom anglais est *shape*.
  - — La commande est `T.shape`.
- ***Nombre d'éléments***.
  - — C'est le nombre total d'éléments nécessaire pour définir un tenseur. Cela correspond au nombre d'emplacements qu'il faut réserver dans la mémoire pour stocker le tenseur. Le nombre d'éléments d'un tenseur est le produit $\ell_1 \times \cdots \times \ell_n$ des longueurs de la taille $(\ell_1, \ldots, \ell_n)$ du tenseur.
  - — Un vecteur de longueur 3 possède 3 éléments, une matrice 3×4 en a 12 et un 4-tenseur de taille $(3, 5, 7, 11)$ en a $3 \times 5 \times 7 \times 11$.
  - — Le nom anglais est *size*.
  - — La commande est `T.size`.

Voici des exemples que l'on a déjà rencontrés :

- Une image 28×28 en niveaux de gris est représentée par un tenseur de taille $(28, 28)$. Sa dimension est 2 et son nombre d'éléments est $784 = 28 \times 28$.
- Une image 32×32 en couleurs est représentée par un tenseur de taille $(32, 32, 3)$. Sa dimension est 3 et son nombre d'éléments est $3072 = 32 \times 32 \times 3$.
- Les données d'apprentissage de la base CIFAR-10 sont composées de 50 000 images couleurs de taille 32×32. L'ensemble des données est donc représenté par un 4-tenseur de taille $(50\,000, 32, 32, 3)$. Un pixel étant codé par un entier sur un octet, il faut réserver $50\,000 \times 32 \times 32 \times 3 = 153.6$ Mo en mémoire.

1.3. Opérations sur les tenseurs

Les tenseurs se comportent comme les vecteurs et les tableaux *numpy* vus dans les chapitres « Python : numpy et matplotlib avec une variable » et « Python : numpy et matplotlib avec deux variables ».
Nous reprenons les exemples définis plus haut.

Opérations sur tous les éléments.

- `V + 3` ajoute 3 à chaque élément de `V`.
- `A ** 2` calcule le carré de chaque élément de `A`.
- `T - 1` retranche 1 à chaque élément de `T`.

Opérations élément par élément entre tenseurs de même taille.

- `A * AA` calcule chaque produit $a_{ij} \times a'_{ij}$ (ici en dimension 2) et n'a rien à voir avec le produit de deux matrices.
- `T + TT` calcule chaque somme $T_{ijk} + T'_{ijk}$ (ici en dimension 3).

Fonctions sur un tenseur.

Certaines fonctions agissent élément par élément d'autre pas.

- `V.mean()` renvoie la moyenne de tous les éléments.
- `A.sum()` renvoie la somme de tous les éléments.
- `np.sqrt(T)` renvoie un tenseur, où chaque élément est obtenu par racine carrée.

1.4. Changer de taille

Changer la taille d'un tenseur tout en conservant le nombre d'éléments est un opération fréquente.

Aplatissement. C'est la mise sous la forme d'un vecteur (tenseur de dimension 1). L'instruction est `T.flatten()` qui pour notre exemple du début de chapitre renvoie le tenseur de taille $(18,)$:

```
[5 1 0 3 3 2 1 1 0 5 1 7 0 0 1 8 1 9]
```

Changement de taille. La méthode `reshape()` transforme un tenseur en changeant sa taille. Par exemple notre tenseur `T` est de taille $(3,2,3)$ et possède 18 éléments.

- `T1 = np.reshape(T,(2,9))` renvoie un tenseur de taille $(2,9)$ contenant les éléments de `T` réorganisés.

```
[[5 1 0 3 3 2 1 1 0]
 [5 1 7 0 0 1 8 1 9]]
```

- `T2 = np.reshape(T,(2,3,3))` renvoie un tenseur de taille $(2,3,3)$.

```
[[[5 1 0]
  [3 3 2]
  [1 1 0]]
 [[5 1 7]
  [0 0 1]
  [8 1 9]]]
```

- `T3 = np.reshape(T,(9,-1))` renvoie un tenseur de taille $(9,2)$. Le « `-1` » indique à *Python* qu'il doit deviner tout seul la taille à calculer (sachant qu'il y a 18 éléments une taille $(9,x)$ impose $x = 2$).

```
[[5 1]
 [0 3]
 [3 2]
 [1 1]
 [0 5]
 [1 7]
 [0 0]
 [1 8]
 [1 9]]
```

Regrouper deux tenseurs. Soient

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

définis par

```
A = np.arange(1,7).reshape((2,3))
B = np.arange(7,13).reshape((2,3))
```

Voici deux commandes illustrant le regroupement de A et B :

```
C1 = np.concatenate((A, B), axis=0)
C2 = np.concatenate((A, B), axis=1)
```

$$C_1 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \qquad C_2 = \begin{pmatrix} 1 & 2 & 3 & 7 & 8 & 9 \\ 4 & 5 & 6 & 10 & 11 & 12 \end{pmatrix}$$

De façon plus générale, toutes les opérations dont vous pourriez avoir besoin sont disponibles, mais il faudra parfois se creuser la tête pour comprendre les mécanismes associés dès que la dimension est plus grande que 3.

2. Tenseurs (avec *tensorflow*)

Avec *tensorflow* les tenseurs sont des objets beaucoup plus puissants que les tableaux *numpy*. Nous ne voyons ici que quelques fonctionnalités, les plus simples.

2.1. Dérivée

Voici comment calculer la dérivée de la fonction $x \mapsto y = x^2$ en $x_0 = 3$.

```
x = tf.Variable(3.)
with tf.GradientTape() as tape:
    y = x**2

tfgradient = tape.gradient(y, [x])
gradient = tfgradient[0].numpy()
print("Dérivée :", gradient)
```

Voici comment calculer la dérivée de la fonction définie par $f(x) = 2\ln(x) + \exp(-x)$ en plusieurs points.

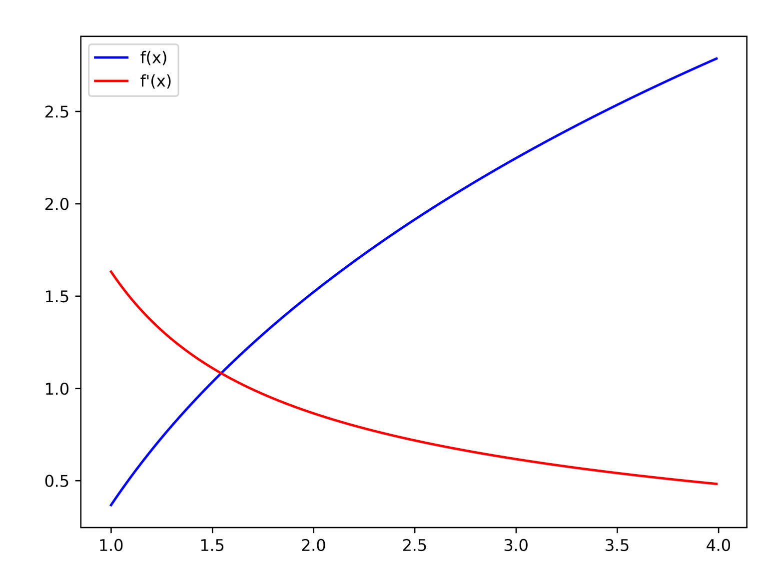
```
def f(x):
    return 2*tf.math.log(x) + tf.math.exp(-x)

x = tf.range(1, 4, 0.5)
with tf.GradientTape() as tape:
    tape.watch(x)
    y = f(x)

tfgradient = tape.gradient(y, x)
gradient = [tfgradient[i].numpy() for i in range(len(tfgradient))]

print("Point x :", x.numpy())
print("Valeur y=f(x) :", y.numpy())
print("Dérivée f'(x) :", gradient)
```

Ici les abscisses x sont les éléments de la liste $[1., 1.5, 2., 2.5, 3., 3.5]$, les valeurs calculées $y = f(x)$ sont $[0.367, 1.034, \ldots]$. Et on obtient la liste des dérivées $f'(x)$: $[1.632, 1.110, \ldots]$.
Avec plus de points, on trace le graphe de f et sa dérivée f' sur l'intervalle $[1, 4]$.



2.2. Gradient

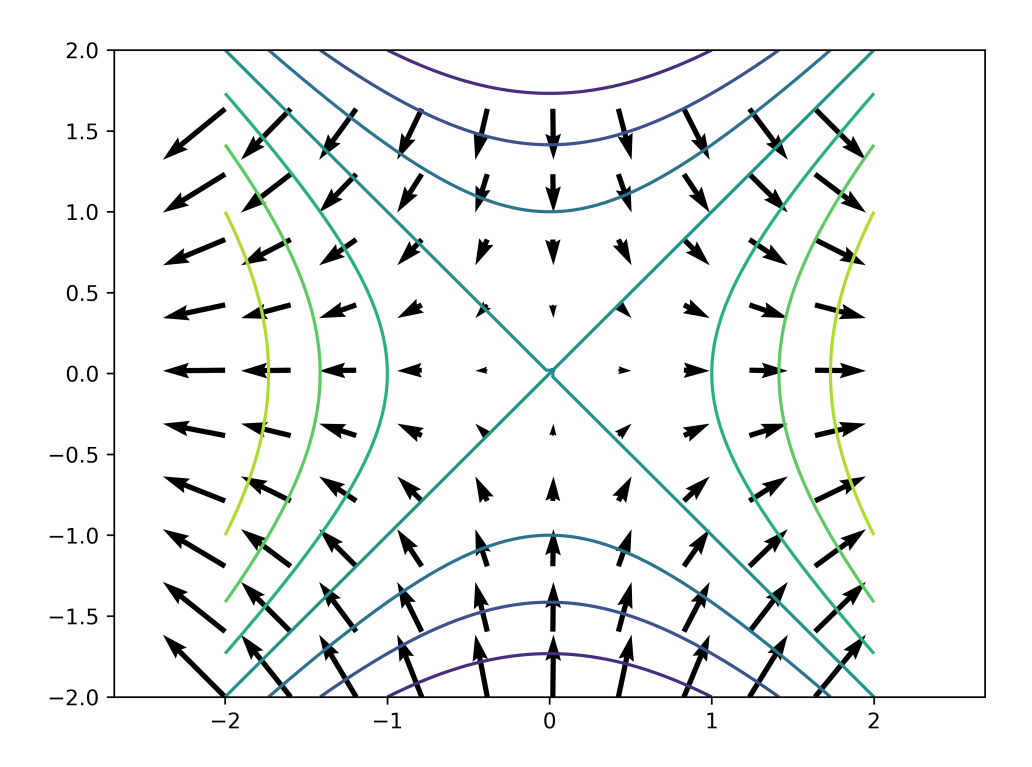
On termine par une fonction particulièrement importante pour ce cours : la fonction gradient.
Voici comment calculer le gradient de $f(x, y) = xy^2$ en $(x_0, y_0) = (2, 3)$.

```
def f(x, y):
    return x * y**2

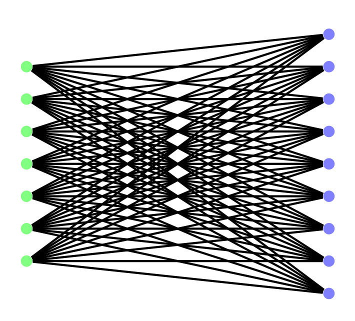
x, y = tf.Variable(2.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(x, y)

tfgradient = tape.gradient(z, [x, y])
gradient = [tfgradient[i].numpy() for i in range(len(tfgradient))]
dfx, dfy = gradient[0], gradient[1]
print("Gradient :", dfx, dfy)
```

On peut ainsi tracer le gradient en chaque point du plan. Ci-dessous le gradient et les lignes de niveau de la fonction $f(x, y) = x^2 - y^2$ (un point-selle).



Troisième partie



compléments

Chapitre 16

Probabilités

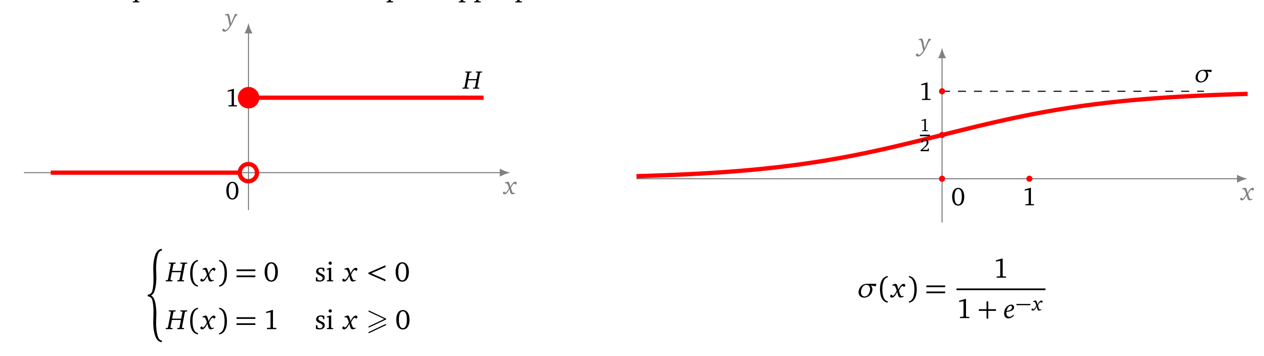
Vidéo ■ partie 16.1. Activation softmax
Vidéo ■ partie 16.2. Fonctions d'erreur
Vidéo ■ partie 16.3. Loi normale

Nous présentons quelques thèmes probabilistes qui interviennent dans les réseaux de neurones.

1. Activation softmax

1.1. La fonction σ

On souhaite pouvoir distinguer une image d'un chat (valeur 0) de celle d'un chien (valeur 1). La première idée serait d'utiliser la fonction marche de Heaviside qui répondrait très clairement à la question, mais nous allons voir que la fonction σ est plus appropriée.



$$\begin{cases} H(x) = 0 & \text{si } x < 0 \\ H(x) = 1 & \text{si } x \geqslant 0 \end{cases} \qquad \sigma(x) = \frac{1}{1+e^{-x}}$$

On rappelle d'abord les qualités de la fonction σ par rapport à la fonction marche de Heaviside :

- La fonction σ est dérivable en tout point avec une dérivée non identiquement nulle, ce qui est important pour la descente de gradient. A contrario la fonction H est de dérivée partout nulle (sauf en 0 où elle n'est pas dérivable).
- La fonction σ prend toutes les valeurs entre 0 et 1, ce qui permet de nuancer le résultat. Par exemple si la valeur est 0.92, on peut affirmer que l'image est plutôt celle d'un chien.

On interprète donc une valeur p entre 0 et 1 renvoyée par σ comme une probabilité, autrement dit un degré de certitude. Évidemment, si l'on souhaite à chaque fois se positionner, on peut répondre : 0 si $\sigma(x) < \frac{1}{2}$ et 1 sinon. Cela revient à composer σ par la fonction $H(x - \frac{1}{2})$.

1.2. Fonction softmax

Comment construire une fonction qui permette de décider non plus entre deux choix possibles, mais parmi un nombre fixé, comme par exemple lorsqu'il faut classer des images en 10 catégories ? Une solution est d'obtenir k valeurs $(x_1, \ldots, x_k)$. On rattache $(x_1, \ldots, x_k)$ à l'une des k catégories données par un vecteur de longueur k :

- $(1,0,0,\ldots,0)$ première catégorie,
- $(0,1,0,\ldots,0)$ deuxième catégorie,
- ...
- $(0,0,0,\ldots,1)$ k-ème et dernière catégorie.

Argmax. La fonction argmax renvoie le rang pour lequel le maximum est atteint. Par exemple si $X = (x_1,x_2,x_3,x_4,x_5) = (3,1,8,6,0)$ alors argmax(X) vaut 3, car le maximum $x_3 = 8$ est atteint au rang 3. Cela correspond donc à la catégorie $(0,0,1,0,0)$.

Softmax. Pour $X = (x_1,x_2,\ldots,x_k)$ on note

$$\sigma_i(X) = \frac{e^{x_i}}{e^{x_1} + e^{x_2} + \cdots + e^{x_k}}.$$

Proposition 1.
Pour chaque $i = 1,\ldots,k$ on a $0 < \sigma_i(X) \leqslant 1$ et

$$\sigma_1(X) + \sigma_2(X) + \cdots + \sigma_k(X) = 1.$$

Comme la somme des $\sigma_i(X)$ vaut 1, on interprète la valeur $\sigma_i(X)$ comme une probabilité p_i. Par exemple avec $X = (3,1,8,6,0)$ alors

$$p_1 = \sigma_1(X) \simeq 0.0059 \quad p_2 = \sigma_2(X) \simeq 0.0008 \quad p_3 \simeq 0.8746 \quad p_4 \simeq 0.1184 \quad p_5 \simeq 0.0003$$

On note que la valeur maximale est obtenue au rang 3. On obtient aussi ce rang 3 en composant par la fonction argmax.

2. Fonctions d'erreur

Prenons un jeu de N données (x_i, y_i) et un réseau de neurones, nous avons pour $i = 1,\ldots,N$:

- la sortie attendue y_i issue des données,
- à comparer avec la sortie $\widetilde{y}_i$ produite par le réseau (calculée comme un $F(x_i)$).

Nous allons voir différentes formules d'erreur. La valeur de l'erreur n'est pas très importante, ce qui est important c'est que cette erreur soit adaptée au problème et tende vers 0 par descente de gradient.

2.1. Erreur quadratique moyenne

L'*erreur quadratique moyenne* calcule une moyenne des carrés des distances entre données et prédictions :

$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - \widetilde{y}_i)^2.$$

Si l'erreur vaut 0, c'est que toutes les valeurs prédites sont exactement les valeurs attendues.

> **Exemple.**
> Avec les données et les prédictions suivantes :
>
> $$(y_i) = [1,2,5,1,3] \qquad \text{et} \qquad (\widetilde{y}_i) = [1.2,1.9,4.5,1.1,2.7],$$
>
> l'erreur quadratique moyenne est $(N = 5)$:
>
> $$E = \frac{1}{5}\big((1-1.2)^2 + (2-1.9)^2 + \cdots\big) = 0.08.$$

2.2. Erreur absolue moyenne

L'*erreur absolue moyenne* calcule une moyenne des distances entre données et prédictions :

$$E = \frac{1}{N}\sum_{i=1}^{N} |y_i - \widetilde{y}_i|.$$

Intuitivement, cette erreur est plus naturelle car elle mesure une distance (et non le carré d'une distance), mais l'usage de la valeur absolue la rend d'utilisation moins aisée que la précédente.

Exemple.
Avec les mêmes données et prédictions que dans l'exemple précédent, cette fois :

$$E = \frac{1}{5}\big(|1-1.2| + |2-1.9| + \cdots\big) = 0.24.$$

2.3. Erreur logarithmique moyenne

L'*erreur logarithmique moyenne* est :

$$E = \frac{1}{N}\sum_{i=1}^{N} \big(\ln(y_i+1) - \ln(\widetilde{y}_i+1)\big)^2.$$

L'usage du logarithme est adapté à des données de différentes tailles. En effet, elle prend en compte de façon identique une petite erreur sur une petite valeur et une grande erreur sur une grande valeur.

Exemple.
Avec les données et prédictions suivantes :

$$(y_i) = [1,5,10,100] \qquad \text{et} \qquad (\widetilde{y}_i) = [2,4,8,105],$$

et $N = 4$, l'erreur est :

$$E = \frac{1}{4}\big((\ln(1+1) - \ln(2+1))^2 + (\ln(5+1) - \ln(4+1))^2 + \cdots + (\ln(100+1) - \ln(105+1))^2\big) \simeq 0.060.$$

Pour illustrer les propos précédents, il est à noter que l'erreur absolue moyenne vaut $E' \simeq 2.25$ et que l'écart entre $y_4 = 100$ et $\widetilde{y}_4 = 105$ contribue à 55% de cette erreur. Tandis que pour l'erreur logarithmique moyenne, l'écart entre $y_4 = 100$ et $\widetilde{y}_4 = 105$ contribue à 15% de l'erreur E. Selon l'origine des données, on peut considérer normal d'avoir une plus grande erreur sur de plus grandes valeurs.

2.4. Entropie croisée binaire

L'*entropie croisée binaire* est adaptée lorsque la réponse attendue est soit $y_i = 0$ soit $y_i = 1$ et que la sortie produite (c'est-à-dire la prédiction) est une probabilité $\widetilde{y}_i$ avec $0 < \widetilde{y}_i < 1$:

$$E = -\frac{1}{N}\sum_{i=1}^{N} \big(y_i \ln(\widetilde{y}_i) + (1-y_i)\ln(1-\widetilde{y}_i)\big).$$

Chaque terme de la somme n'est en fait constitué que d'un seul élément : si $y_i = 1$ c'est $\ln(\widetilde{y}_i)$ et si $y_i = 0$ c'est $\ln(1-\widetilde{y}_i)$.

Exemple.
Avec les données et prédictions suivantes :

$$(y_i) = [1,0,1,1,0,1] \qquad \text{et} \qquad (\widetilde{y}_i) = [0.9,0.2,0.8,1.0,0.1,0.7].$$

et $N = 6$, l'entropie croisée binaire est :

$$E = -\frac{1}{6}\Big(\ln(0.9) + \ln(1-0.2) + \ln(0.8) + \ln(1.0) + \ln(1-0.1) + \ln(0.7)\Big) \simeq 0.17.$$

Les couleurs sont à mettre en correspondance avec les valeurs des données :

$$[1, 0, 1, 1, 0, 1].$$

2.5. Entropie croisée pour catégories

L'*entropie croisée pour catégories* est adaptée lorsque la réponse attendue est un vecteur du type $y_i = (0, 0, \ldots, 1, \ldots, 0)$ (avec un seul 1 à la place qui désigne le rang de la catégorie attendue) et que la sortie produite par la prédiction est une liste de probabilités $\widetilde{y}_i = (p_1, p_2, \ldots)$ (voir l'exemple de la reconnaissance de chiffres).

3. Normalisation

3.1. Motivation

Les données brutes qui servent à l'apprentissage peuvent avoir des valeurs diverses. Afin de standardiser les procédures, il est préférable de ramener les valeurs des données dans un intervalle déterminé par exemple $[0, 1]$ ou bien $[-1, 1]$. C'est plus efficace car au départ de l'apprentissage les valeurs initiales des poids du réseau sont fixées indépendamment des données, par exemple au hasard entre $[-1, 1]$. Cela permet également de partir de poids qui ont été calculés pour une autre série de données au lieu de valeurs aléatoires.

Prenons l'exemple d'images en niveau de gris, pour lesquelles chaque pixel est codé par un entier n_{ij} entre 0 et 255. On « normalise » l'image en remplaçant la matrice des pixels (n_{ij}) par la matrice $(n_{ij}/255)$. Ainsi les valeurs sont maintenant des réels entre 0 et 1.

Plus généralement, si les données sont comprises entre $y_{\min}$ et $y_{\max}$ alors on redéfinit les données à l'aide de la fonction affine suivante dont l'ensemble d'arrivée est l'intervalle $[0, 1]$:

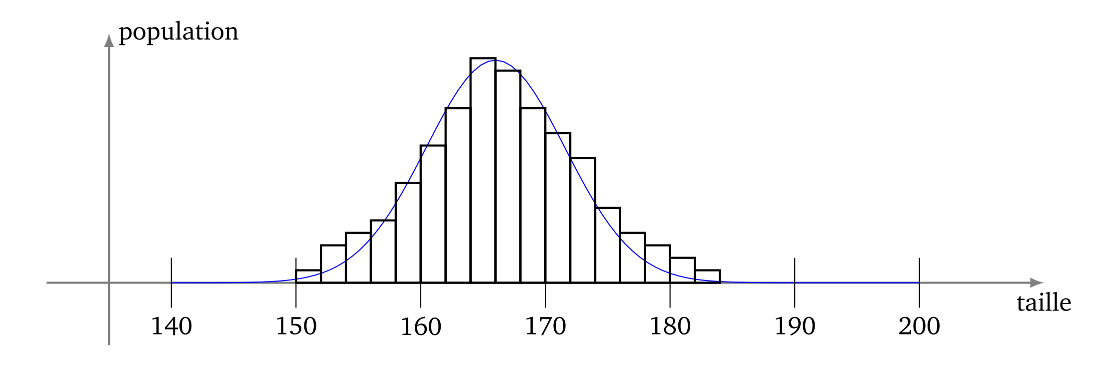
$$y \mapsto \frac{y - y_{\min}}{y_{\max} - y_{\min}}.$$

Pour les ramener dans l'intervalle $[a, b]$, on utiliserait la fonction :

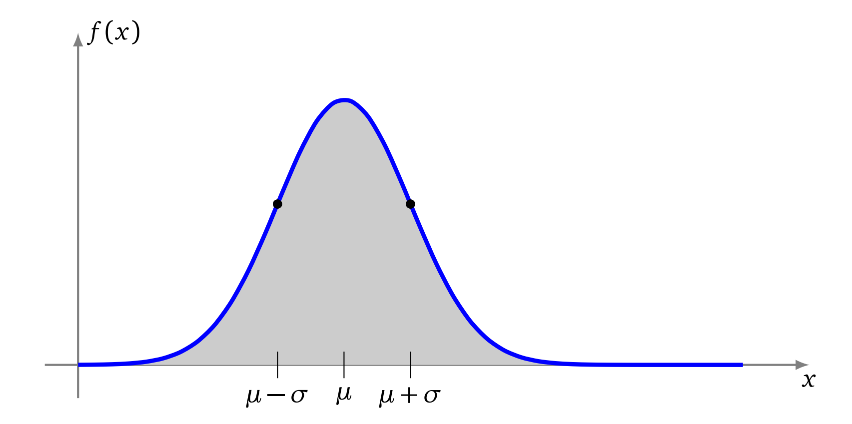
$$y \mapsto \frac{y - y_{\min}}{y_{\max} - y_{\min}}(a - b) + b.$$

3.2. Loi de Gauss

Cependant pour certaines données, nous n'avons pas connaissance de leurs valeurs extrêmes. Prenons l'exemple de la taille de personnes. Voici la répartition du nombre de femmes en fonction de leur taille.



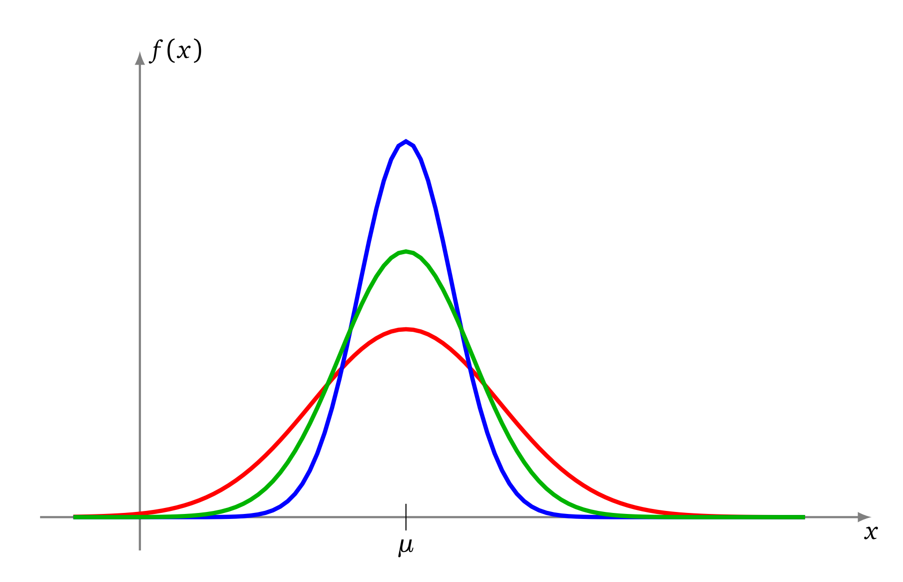
Une telle répartition se fait suivant la « loi de Gauss » que l'on nomme aussi « courbe en cloche ». Elle dépend de deux paramètres : l'*espérance* μ et l'*écart-type* σ.



L'aire sous la courbe vaut 1, la courbe est symétrique par rapport à l'axe $(x=\mu)$ et elle possède deux points d'inflexion en $x=\mu-\sigma$ et $x=\mu+\sigma$. L'équation est la suivante :

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

La variable μ définit l'axe de symétrie, tandis que la variable σ détermine l'étalement de la courbe. Voici différentes courbes pour la même valeur de μ mais pour différentes valeurs de σ.



3.3. Espérance, écart-type

On se donne une série de données $(x_i)_{1\leqslant i\leqslant n}$. On souhaite retrouver les paramètres μ et σ correspondant à cette série.

- L'*espérance* est la moyenne des données et se calcule par :

$$\mu=\frac{1}{n}\sum_{i=1}^{n}x_i.$$

- On note $\text{Var}(x)$ la *variance* des $(x_i)_{1\leqslant i\leqslant n}$:

$$\text{Var}(x)=\frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2.$$

  La variance mesure l'écart des valeurs avec la moyenne.
- L'*écart-type* est la racine carrée de la variance :

$$\sigma=\sqrt{\text{Var}(x)}.$$

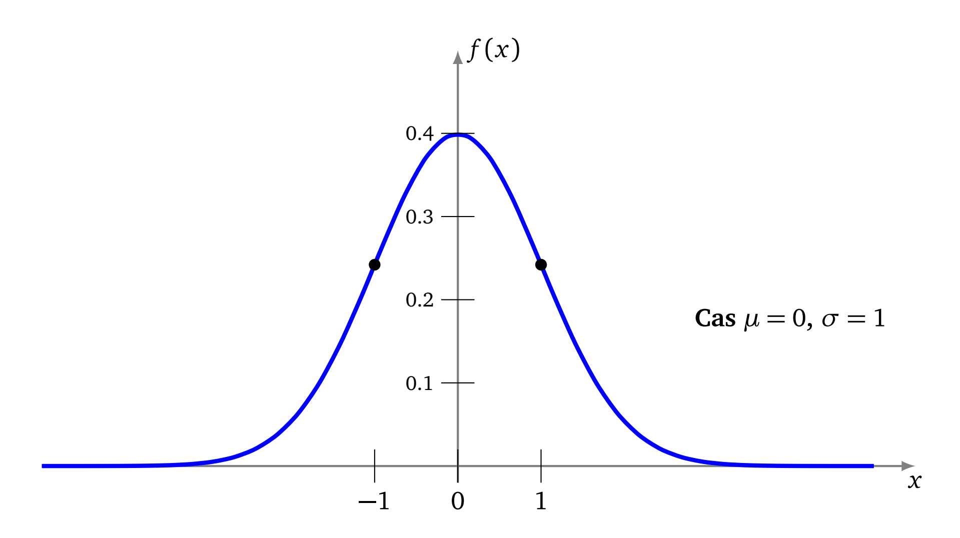
La *loi normale* ou *loi de Gauss* $\mathcal{N}(\mu,\sigma)$ est alors la loi associée à la densité de probabilité :

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

3.4. Loi normale centrée et réduite

On souhaite normaliser nos données en se ramenant à une loi normale $\mathcal{N}(0,1)$ qui est donc centrée ($\mu=0$) et réduite ($\sigma=1$). La *loi de Gauss centrée réduite* est donc définie par :

$$f(x)=\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}.$$



Si les données $(x_i)_{1\leqslant i\leqslant n}$ ont pour espérance μ et écart-type σ, alors les données $(x'_i)_{1\leqslant i\leqslant n}$ définies par

$$x'_i=\frac{x_i-\mu}{\sigma}$$

ont pour espérance $\mu'=0$ et écart-type $\sigma'=1$.

3.5. Exemple

Voici un échantillon de tailles d'hommes (en cm) :

$$[172,165,187,181,167,184,168,174,180,186].$$

Pour déterminer une loi de Gauss à partir de cet échantillon, on calcule l'espérance :

$$\mu=\frac{172+165+\cdots+186}{10}=176.4$$

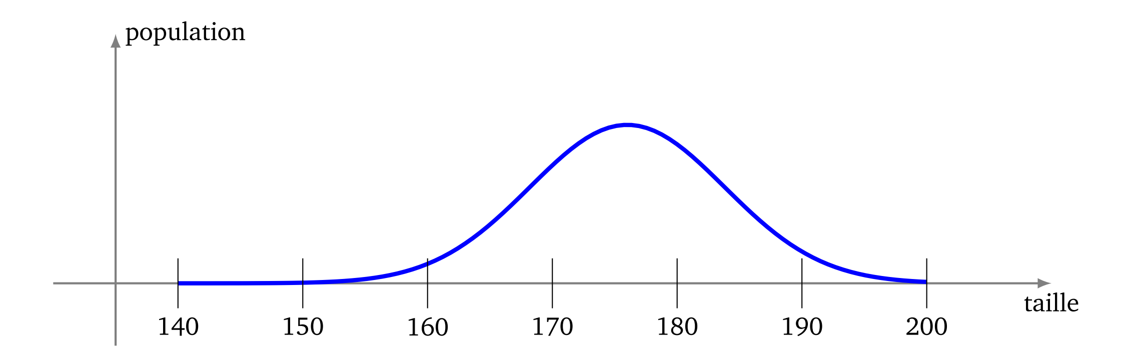
puis la variance :

$$\mathrm{Var}=\frac{(172-\mu)^2+(165-\mu)^2+\cdots+(186-\mu)^2}{10}=61.04$$

qui donne l'écart-type

$$\sigma=\sqrt{\mathrm{Var}}\simeq 7.8.$$

Ainsi on modélise la taille des hommes par une loi normale $\mathcal{N}(\mu,\sigma)$ avec $\mu\simeq 176$ et $\sigma\simeq 7.8$.



On en déduit la répartition attendue des hommes en fonction de leur taille, par exemple 95% des hommes ont une taille comprise entre $\mu - 2\sigma$ et $\mu + 2\sigma$, c'est-à-dire dans l'intervalle $[161, 192]$.
Pour obtenir une loi normale centrée réduite on utilise la transformation :

$$x'_i = \frac{x_i - \mu}{\sigma}$$

où x_i est la hauteur (en cm) et x'_i est la donnée transformée (sans unité ni signification physique). Ce qui nous donne des données transformées x'_i (arrondies) :

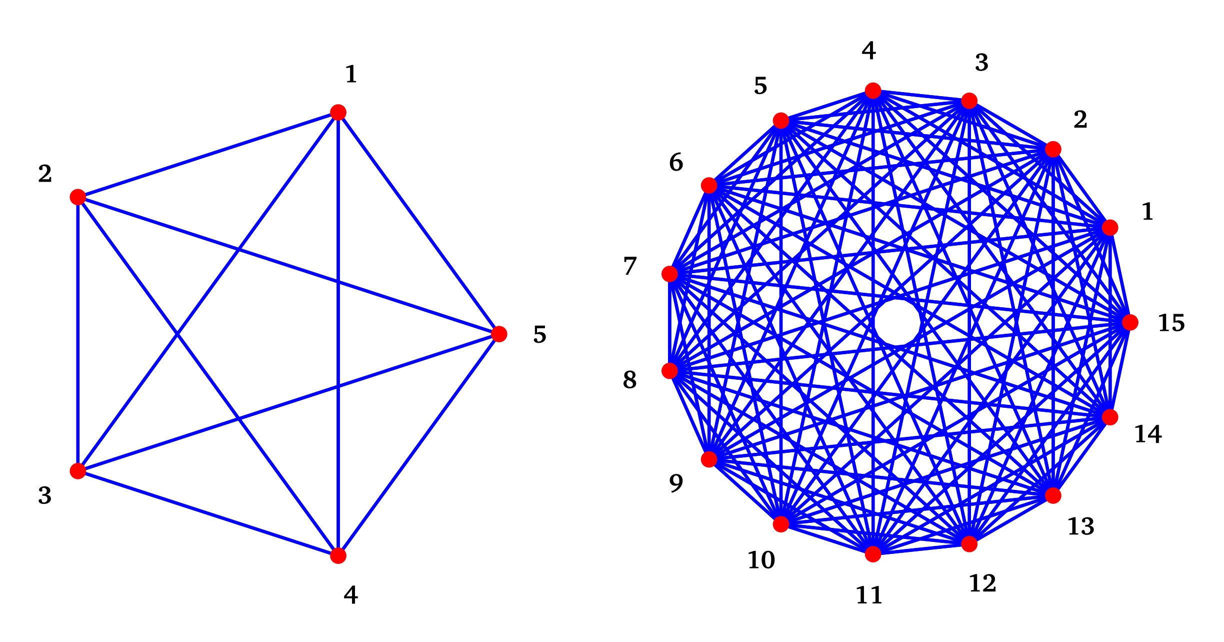
$$[-0.56, -1.46, 1.36, 0.59, -1.20, 0.97, -1.07, -0.31, 0.46, 1.23].$$

4. Dropout

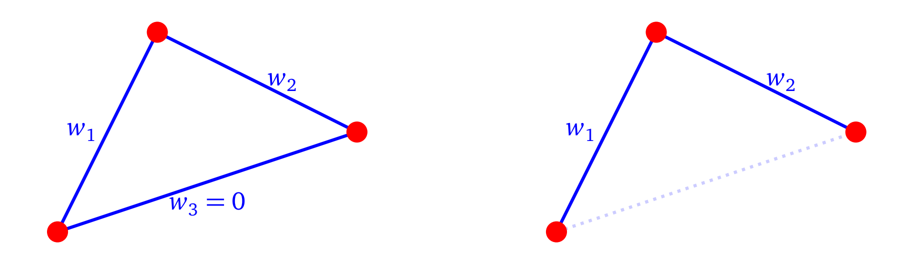
Le dropout est une technique qui simule différentes configurations de liens entre les neurones et limite le sur-apprentissage.

4.1. Motivation

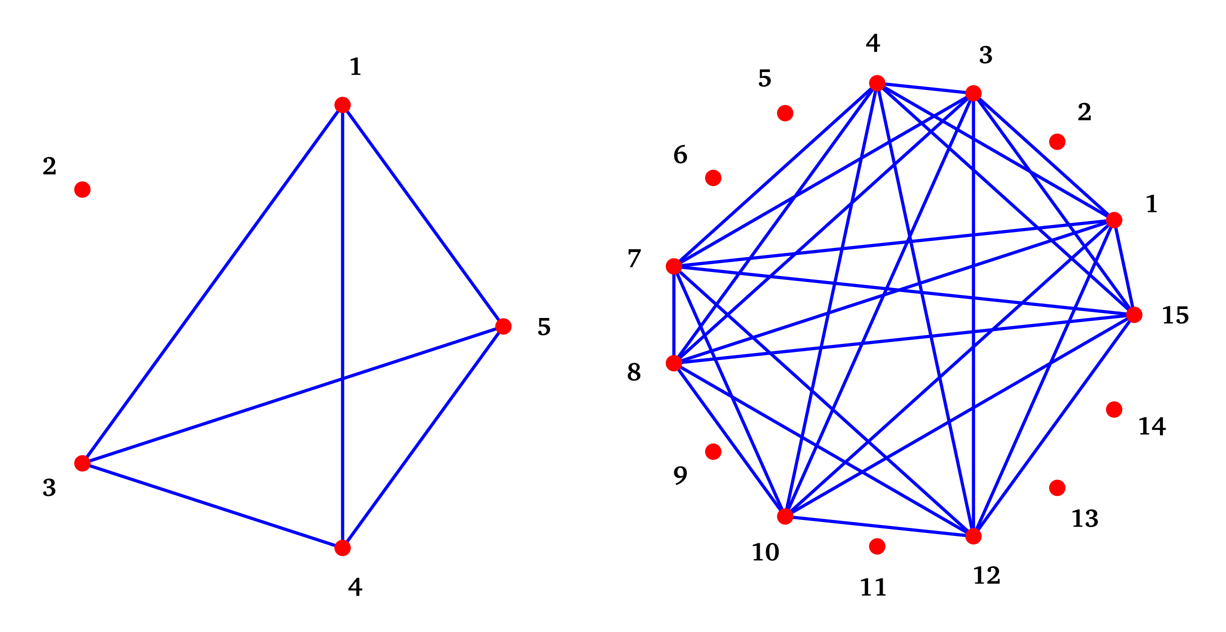
Imaginons n neurones. Quelle est la meilleure architecture pour les relier entre eux ? Bien sûr la réponse dépend du problème, ainsi on ne peut pas le savoir à l'avance.
Une première façon de faire est de relier tous les neurones entre eux.



Lors de l'apprentissage certains poids vont peut-être devenir nuls (ou presque) ce qui revient à faire disparaître l'arête reliant deux neurones.



Une autre façon de procéder est de désactiver dès le départ certains neurones avant de commencer l'apprentissage. Cela revient à imposer un poids 0 immuable au cours de l'apprentissage à toutes les arêtes reliées à ces neurones. Mais quelle configuration choisir ? Avec n neurones, il existe $N = 2^n$ configurations possibles. Voici deux exemples :



Pour $n = 5$ cela fait $N = 32$ configurations, pour $n = 15$ cela donne plus de 32 768 possibilités ! Pour chacune de ces configurations, il faudrait appliquer la descente de gradient. C'est beaucoup trop de travail, même pour un ordinateur.

Voici l'idée du dropout (avec paramètre $p = 0.5$). On part d'un réseau de n neurones, tous reliés les uns aux autres. Avant la première étape d'apprentissage, on décide de désactiver certains neurones. Cette décision est prise au hasard. Pour chaque neurone on lance une pièce de monnaie, si c'est « pile » on conserve le neurone, si c'est « face » on le désactive. Ensuite on effectue une étape de la descente de gradient, avec seulement une partie de nos neurones activés. Avant la deuxième étape de la descente de gradient, on reprend notre pièce et on choisit au hasard les neurones à désactiver, etc.

4.2. Loi de Bernoulli

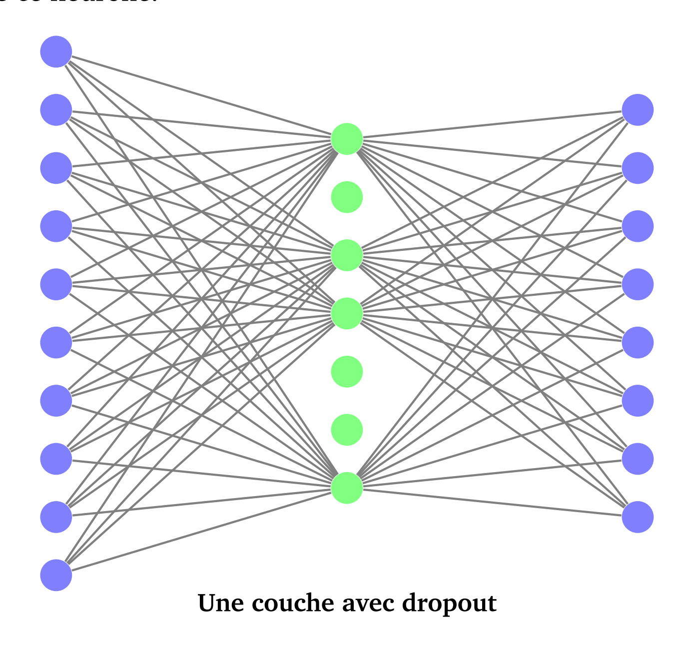
Une variable aléatoire suit la ***loi de Bernoulli de paramètre p*** si le résultat est 1 avec une probabilité p et 0 avec une probabilité $1 - p$.

Par exemple, le lancer d'une pièce de monnaie non truquée suit une loi de Bernoulli de paramètre $p = 0.5$.

Désactiver les neurones suivant un loi de Bernoulli de paramètre $p = 0.8$ signifie que chaque neurone à 8 chances sur 10 d'être conservé (et donc 2 chances sur 10 d'être désactivé). Sur un grand nombre de neurones, on peut estimer qu'environ 80% sont conservés, mais il peut y avoir des tirages exceptionnels pour lesquels ce n'est pas le cas.

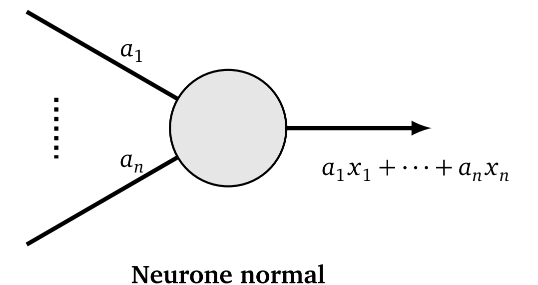
4.3. Dropout

Revenons à nos réseaux de neurones ou plus exactement à une couche de n neurones. Appliquer un ***dropout*** à cette couche, c'est désactiver chaque neurone suivant une loi de Bernoulli de paramètre p, où $0 < p \leqslant 1$ est un réel fixé. On rappelle que désactiver un neurone signifie mettre les poids à 0 pour toutes les arêtes entrantes et sortantes de ce neurone.



Une couche avec dropout

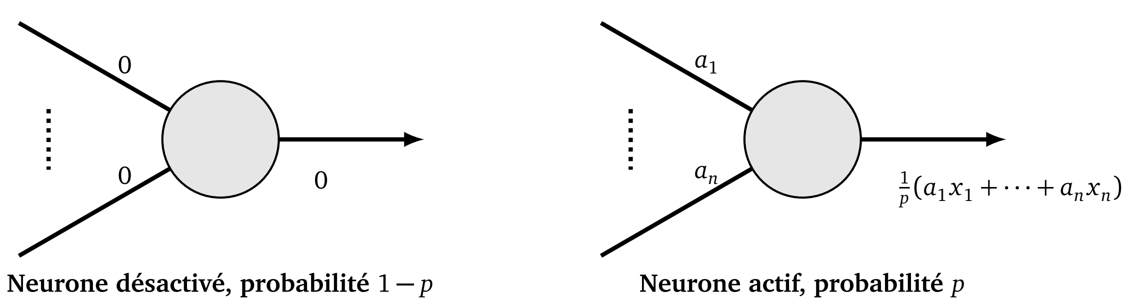
Neurone « normal ».
Partons d'un neurone classique (sans fonction d'activation) et voyons comment le modifier pour l'apprentissage avec dropout.



Neurone normal

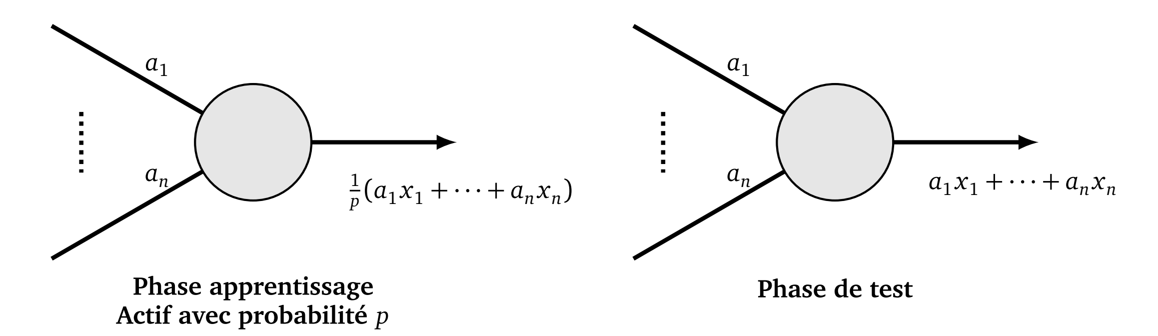
Apprentissage. Lors de la phase d'apprentissage, si le neurone est désactivé (avec une probabilité $1-p$) alors tous les poids sont nuls, et bien sûr la valeur renvoyée est nulle. Par contre si le neurone est activé (avec une probabilité p), alors il se comporte comme un neurone « presque normal », il faut juste pondérer la sortie d'un facteur $\frac{1}{p}$ (comme il y a moins de neurones chaque neurone doit contribuer de façon plus importante).





Neurone désactivé, probabilité $1-p$ **Neurone actif, probabilité** p

Test. Lors de la phase de test, tous les neurones sont actifs. La sortie est calculée comme un neurone normal.


a_1
a_n
$\frac{1}{p}(a_1x_1 + \cdots + a_nx_n)$
Phase apprentissage
Actif avec probabilité p
a_1
a_n
$a_1x_1 + \cdots + a_nx_n$
Phase de test

Annexe

On regroupe ici quelques informations diverses et variées.

1. Une brève histoire des réseaux de neurones

- **Neurone en biologie.** Entre 1840 et 1900 on découvre que la cellule est l'élément fondamental du vivant ; par observation, Santiago Ramón y Cajal et d'autres mettent en évidence que les cellules du système nerveux forment un réseau : les neurones. On sait maintenant que le cerveau humain contient entre 80 et 100 milliards de neurones, mais nous avons aussi 500 millions de neurones dans le ventre (soit autant que dans le cerveau d'un chien). Un neurone est composé d'un élément central prolongé par un axone au bout duquel se trouvent les synapses. Les neurones sont organisés en réseau, un neurone étant en moyenne relié à 10 000 autres neurones et transmet ainsi un signal électrique à tous ses neurones voisins.
- **De la biologie vers l'informatique.** À la fin des années 1940, Donald Hebb émet l'hypothèse que lors de l'apprentissage certaines connexions synaptiques se renforcent et que ce renforcement persiste. Un modèle mathématique est déduit par Frank Rosenblatt : le perceptron (1958). L'algorithme est mis en œuvre dans une machine dédiée à la reconnaissance d'images. Le perceptron, qui correspond à un réseau formé d'un seul neurone, montre très vite ses limites à la fois théoriques (puisqu'il ne peut réaliser le « ou exclusif ») et aussi en termes de résultats pratiques. Ces deux problèmes sont résolus en considérant des réseaux à plusieurs couches, mais les calculs à mener sont trop longs à la fois en raison de la puissance des ordinateurs de l'époque et des algorithmes utilisés.
- **Rétropropagation.** Un progrès important est fait grâce l'algorithme de rétropropagation (par Rumelhart, Hinton et Williams en 1986) qui permet un calcul efficace des poids des neurones. On conserve encore aujourd'hui ce principe : des calculs essentiellement numériques, avec un minimum de calculs symboliques au niveau de chaque neurone (afin de calculer la dérivée), le tout avec un grand nombre d'itérations. Des améliorations se succèdent : pooling, dropout, calculs en parallèle, meilleures descentes de gradient.
- **Hiver.** Cependant les attentes de l'intelligence artificielle (peut être le nom a-t-il été mal choisi ?) sont largement déçues car ses applications restent limitées. L'intérêt des scientifiques diminue drastiquement. De 1980 à 2000 on parle de l'hiver de l'intelligence artificielle.
- ***Deep learning*.** À partir des années 2000 et surtout après 2010 les réseaux de neurones font des progrès fulgurants grâce à l'apprentissage profond. Yann Le Cun démontre l'efficacité des couches de convolution pour la reconnaissance des chiffres. On réalise et entraîne alors des réseaux ayant de plus en plus de couches grâce à des progrès matériels (par exemple le calcul sur les processeurs graphiques GPU) mais surtout grâce aux couches de convolution qui extraient des caractéristiques abstraites des images.
- **Présent et avenir.** Les réseaux de neurones s'appliquent à de nombreux domaines : la reconnaissance d'images (par exemple la détection de cancer sur une radiographie), les transports (par exemple la

conduite autonome des voitures), les jeux (les ordinateurs battent les champions du monde d'échecs, de go et des jeux vidéos les plus complexes), l'écriture (classement, résumé, traduction)... Il persiste cependant une certaine méfiance vis à vis des décisions prises par une machine (sentence de justice, diagnostic médical, publicité ciblée). Une meilleure compréhension du fonctionnement des réseaux de neurones par tous est donc indispensable !

2. Références

- *tensorflow* est développé par *Google*. De nombreux tutoriels sont disponibles pour les débutants :
  tensorflow.org
- *keras* est maintenant intégré à *tensorflow* et facilite sa prise en main. Il a été développé par François Cholet auteur du livre *Deep learning with Python* (Manning publications, 2017). Il y a de nombreux exemples d'application :
  keras.io
- Le livre *Deep learning/L'apprentissage profond* par Goodfellow, Bengio, Courville contient des concepts plus avancés. Il est disponible en anglais et en français. Une version gratuite est consultable ici :
  deeplearningbook.org
- Vous pouvez récupérer l'intégralité des codes *Python* ainsi que tous les fichiers sources sur la page *GitHub* d'Exo7 : « GitHub : Deepmath ».
  En particulier vous trouvez sur ce site le module `keras_facile` qui vous aide à définir facilement des poids pour un réseau simple.
- Un peu de pub pour les livres Exo7 :
  — *Algèbre* et *Analyse* pour toutes les notions de mathématiques niveau première année d'études supérieures,
  — *Python au lycée* (tome 1 et tome 2) pour apprendre la programmation.
  Ils sont disponibles gratuitement en téléchargement :
  exo7.emath.fr et GitHub : exo7math
  et aussi en vente à prix coûtant sur amazon.fr en version imprimée.

3. Ce qu'il faut pour utiliser tensorflow/keras

Les activités de ce livre sont écrite pour *Python*, version 3. Cependant il faut installer un certain nombre de modules complémentaires qui ne sont pas nécessairement présents par défaut.

- `tensorflow` qui contient le sous-module `keras`,
- `numpy` pour les tableaux,
- `matplotlib` pour l'affichage de graphiques,
- `scipy` pour la convolution,
- `ioimage` pour gérer la lecture et l'écriture d'images.

Un module s'installe simplement par :

```
pip install mon_module
```

Il existe des distributions *Python* (du type `conda`) qui contiennent les principaux modules scientifiques et permettent aussi de créer des « environnements » afin de gérer plusieurs versions de *Python* et de ses modules.

Pour ceux qui souhaitent ne rien installer, il est possible d'utiliser *tensorflow* en ligne :

Google Colab

4. Lexique français/anglais

La plupart des références à propos des réseaux de neurones sont en anglais. Voici donc un petit lexique français/anglais des termes principaux.

Réseau de neurones. *Neural network.*

- neurone artificiel/*artificial neuron*
- poids/*weights*, biais/*bias*
- fonction d'activation/*activation function*
- fonction marche de Heaviside/*Heaviside step function*
- fonction sigma σ ou sigmoïde/*sigmoid function*
- fonction d'erreur/*loss function* ou *cost function*
- entrée/*input*, sortie/*output*
- couches du réseau/*layers*
- réseau de neurones avec convolution/*convolutional neural network* abrégé en *cnn*
- un motif de convolution s'appelle aussi noyau/*kernel* ou filtre/*filter* ou masque/*mask*
- *dropout* se traduit par abandon ou décrochage
- *pooling* se traduit par regroupement de termes

Mathématiques.

- dérivée partielle/*partial derivative*
- gradient/*gradient* mais la notation anglo-saxonne de grad $f(x, y)$ est $\nabla f(x, y)$
- régression linéaire/*linear regression*
- tensor/*tensor*, taille/*shape*, nombre d'éléments/*size*

Descente de gradient. *Gradient descent.*

- descente de gradient (classique)/*(batch) gradient descent*
- descente de gradient stochastique/*stochastic gradient descent* abrégée en *sgd*
- descente de gradient par lots/*mini-batch gradient descent*
- pas δ/*learning rate*
- erreur quadratique moyenne/*minimal squared error* abrégée en *mse*
- moment/*momentum*
- époque/*epoch*

5. L'image de couverture

L'image de couverture du livre est générée automatiquement grâce à un réseau de neurones et la rétropropagation. Au départ il faut deux images :

- une image dont on veut conserver le contenu : ici la statue de la liberté,
- et une image dont on veut conserver le style : ici un tableau de Van Gogh.

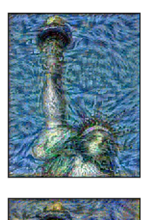
L'image obtenue possède le contenu de la première et le style de la seconde !

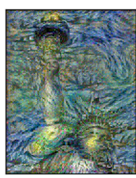


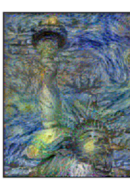
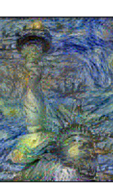


Références :

- Statue de la liberté (photographie par George Hodan, licence CC0, domaine public).
- Peinture de Vincent Van Gogh, *La nuit étoilée*, 1889.
- Le programme et les explications pour *tensorflow* : *Neural Style Transfer* avec TensorFlow d'après des travaux de Gatys, Ecker et Bethge.

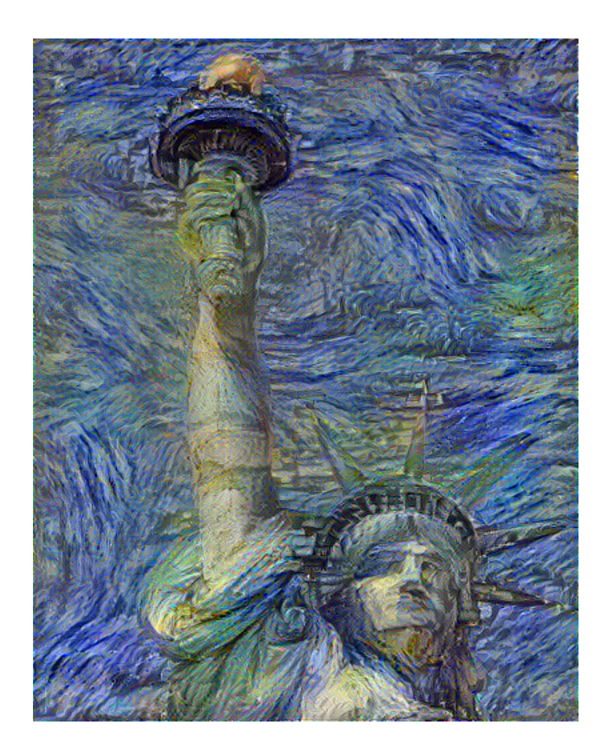



Plus en détails :

- le réseau de neurones, nommé « VGG19 », est un réseau à 19 couches. On ne va pas calculer les poids, mais on utilise des poids qui ont déjà été calculés pour reconnaître les images de la base *ImageNet*.
- Dans ce réseau bien paramétré certaines couches se concentrent sur le contenu (contour, lignes, formes...) et d'autres se concentrent sur le style (couleur, flou...).
- En identifiant ces couches, on construit par itérations une image qui conserve le contenu de la première image et prend peu à peu le style de la seconde.
- Ces itérations se font par rétropropagation et descente de gradient selon une fonction d'erreur qui est la somme d'une fonction d'erreur associée au contenu et d'une fonction d'erreur associée au style.



Remerciements

Nous remercions Michel Bodin pour sa relecture. Merci à Kroum Tzanev pour ses figures de convolutions et à Frédéric Sanchez pour ses remarques pertinentes.

- Page du projet : « Page GitHub : Deepmath ».
- Vidéos « Youtube : Deepmath ».
- Vous pouvez récupérer l'intégralité des codes *Python* ainsi que tous les fichiers sources sur la page *GitHub* d'Exo7 : « GitHub : Deepmath ».
- Autres ressources : exo7.emath.fr.



Index